Kristiania

Search-Based Automated Mocking for System-Level Testing and Vulnerability Detection for RESTful APIs

Susruthan Seran

PhD Dissertations Kristiania 2026:1

Search-Based Automated Mocking for System-Level Testing and Vulnerability Detection for RESTful APIs

Susruthan Seran

Thesis submitted for the degree of Philosophiae Doctor (PhD) in

Applied Information Technology

Kristiania University of Applied Sciences

Fall 2025

PhD Dissertations Kristiania no.1, 2026
School of Doctoral Studies
Kristiania University of Applied Sciences
Oslo, Norway

ISSN: 3084-1488 (print)
ISSN: 3084-147X (online)

ISBN: 978-82-93953-14-2 (print)
ISBN: 978-82-93953-13-5 (online)

DOI: https://doi.org/10.57933/9qwp-az28

Printed at Byråservice

Acknowledgments

This journey would not have been possible without the support, guidance, and encouragement of many individuals. My deepest gratitude goes to everyone who contributed, both directly and indirectly, to the completion of this research and thesis.

I sincerely thank my supervisors, Prof. Andrea Arcuri and Dr. Guru Prasad, and my previous second supervisors, Dr. Man Zhang and Dr. Onur Duman, for their unwavering support, invaluable mentorship, and intellectual guidance throughout my doctoral studies. Their patience, insightful feedback, and profound knowledge of software engineering were instrumental in shaping this research and in my development as a researcher. I am deeply grateful for the trust they placed in me and for their commitment to my success.

I would like to thank all my colleagues at The School of Economics, Innovation and Technology, and The School of Doctoral Studies at Kristiania University of Applied Sciences for their invaluable support and assistance.

I am immensely grateful to the European Research Council for the financial support that made this research possible.

Finally, and most importantly, I dedicate this work to my family and friends. Thank you for believing in me and for reminding me to take breaks when I needed them the most. This achievement is as much yours as it is mine.

Abstract

This thesis addresses key challenges in **automated software testing** for modern, **distributed software systems**, specifically focusing on applications that rely heavily on **external web services** (e.g., microservices, third-party APIs). The shift from monolithic to complex, interconnected architectures has made comprehensive system-level testing increasingly difficult, particularly when handling external communications.

We present a suite of novel, search-based fuzzing techniques (i.e., white-box and black-box) to reduce manual intervention in testing these systems. Our primary contributions are:

1. **Automated Mocking:** We introduce novel search-based white-box fuzzing techniques for **automatically generating mock external web services** for JVM-based applications, eliminating the need for developers to hand-craft mock services and enabling robust testing of dependency interactions.

2. **Automated Schema Inference:** We solve the challenge of generating well-formed, complex inputs (such as **JSON documents**) for white-box fuzzing by developing techniques to **infer schemas in a fully automated manner**, thus supporting zero manual intervention.

3. **Automated SSRF Detection:** Recognizing that external communications introduce security threats, we present novel techniques to automatically **detect and exploit Server-Side Request Forgery (SSRF)** vulnerabilities in both white-box and black-box fuzzing contexts.

These novel techniques are implemented as an extension to the open-source fuzzer EvoMaster and evaluated using multiple open-source and industrial case studies, including those with known Common Vulnerabilities and Exposures (CVEs). Ultimately, this research provides a significant step toward **fully automated, system-level testing** by generating self-contained test suites with dynamic mocking capabilities and SSRF detection.

Sammendrag

Denne avhandlingen tar for seg sentrale utfordringer innen **automatisert programvaretesting** for moderne, **distribuerte programvaresystemer**, med spesielt fokus på applikasjoner som er sterkt avhengige av **eksterne nettjenester** (f.eks. mikrotjenester, tredjeparts-APIer).

Overgangen fra monolittiske til komplekse, sammenkoblede arkitekturer har gjort omfattende systemtesting stadig vanskeligere, spesielt når det gjelder håndtering av ekstern kommunikasjon.

Vi presenterer en rekke nye, søkbaserte fuzzing-teknikker (dvs. white-box og black-box) for å redusere manuelt arbeid i testingen av disse systemene. Våre hovedbidrag er:

1. **Automatisert Mocking:** Vi introduserer nye søkbaserte white-box fuzzing-teknikker for **automatisk generering av falske (mock) eksterne nettjenester** for JVM-baserte applikasjoner. Dette eliminerer behovet for at utviklere må håndlage falske tjenester og muliggjør robust testing av avhengighetsinteraksjoner.

2. **Automatisert Skjemainferens:** Vi løser utfordringen med å generere velfungerende, komplekse inndata (som **JSON-dokumenter**) for white-box fuzzing ved å utvikle teknikker for å **utlede skjemaer på en fullstendig automatisert måte**, og dermed støtte null manuell inngripen.

3. **Automatisert SSRF-deteksjon:** Med erkjennelsen av at ekstern kommunikasjon introduserer sikkerhetstrusler, presenterer vi nye teknikker for automatisk å **oppdage og utnytte Server-Side Request Forgery (SSRF)**-sårbarheter i både white-box og black-box fuzzing-sammenhenger.

Disse nye teknikkene er implementert som en utvidelse til åpen kildekode-fuzzeren EvoMaster og er evaluert ved bruk av flere åpen kildekode- og industrielle case-studier, inkludert de med kjente Common Vulnerabilities and Exposures (CVEs).

Til syvende og sist representerer denne forskningen et betydelig skritt mot **fullt automatisert systemtesting** ved å generere selvstendige testsuiter med dynamiske mocking-funksjoner og SSRF-deteksjon.

List of Articles

The following papers are included in this thesis:

1. **Susruthan Seran**, Man Zhang, and Andrea Arcuri. **Search-based mock generation of external web service interactions**. In *International Symposium on Search Based Software Engineering*, pages 52-66. Springer, 2023. https://doi.org/10.1007/978-3-031-48796-5_4

2. **Susruthan Seran**, Man Zhang, Onur Duman, and Andrea Arcuri. **Handling web service interactions in fuzzing with search-based mock-generation**. *ACM Transactions on Software Engineering and Methodology*, 2025. https://doi.org/10.1145/3731558

3. **Susruthan Seran**, Onur Duman, and Andrea Arcuri. **Multi-phase taint analysis for json inference in search-based fuzzing**. In *The 18th Intl. Workshop on Search-Based and Fuzz Testing*. IEEE, 2025. https://doi.org/10.1109/SBFT66712.2025.00015

4. **Susruthan Seran**, Guru Prasad Bhandari, and Andrea Arcuri. **Detecting Server-Side Request Forgery (SSRF) Vulnerabilities In REST API Fuzz Testing**. In *pending*. pending, 2026. [Unpublished manuscript]

The following articles are not included in this thesis:

- Andrea Arcuri, Man Zhang, Amid Golmohammadi, Asma Belhadi, Juan P Galeotti, Bogdan Marculescu, and **Susruthan Seran**. **EMB: A curated corpus of web/enterprise applications and library support for software testing research**. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 433-442. IEEE, 2023. https://doi.org/10.1109/ICST57152.2023.00047

- Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan Marculescu, Amid Golmohammadi, Juan Pablo Galeotti, and **Susruthan Seran**. **Building an open-source system test generation tool: lessons learned and empirical analyses with evomaster**. *Software Quality Journal*, pages 1-44, 2023. https://doi.org/10.1007/s11219-023-09620-w

- **Seran, Susruthan**. **Search-Based Security Testing of Enterprise Microservices**. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 463-465. IEEE, 2024. https://doi.org/10.1109/ICST60714.2024.00056

- Arcuri, Andrea and Zhang, Man and **Seran, Susruthan** and Galeotti, Juan Pablo and Golmohammadi, Amid and Duman, Onur and Aldasoro, Agustina and Ghianni, Hernan. **Tool report: EvoMaster—black and white box search-based fuzzing for REST, GraphQL and RPC APIs**. In *Automated Software Engineering*, pages 1–11, 2025. https://doi.org/10.1007/s10515-024-00478-1

Contents

Part I

Thesis

Chapter 1

Summary

1.1 Introduction

Software has become an integral component of daily life, from coffee machines to satellites in space. Such an explosion in adaptation has drastically shaped the way we build software. Software architectural patterns have shifted from monoliths towards interconnected complex distributed software systems, such as microservices [49].

Simultaneously, since the first software test performed in the early 1950s, software testing has evolved significantly, transitioning from meticulously reviewing program instructions line-by-line to the current reality of automated software testing. Modern-day software systems are connected to each other to execute one common goal. Testing software with multiple external communications, such as microservice-based enterprise systems, presents significant challenges [47, 49]. While microservice architecture simplifies the development process, it introduces additional complexity to system-level software testing [49]. Furthermore, testing a software system with external service dependencies in isolation becomes increasingly challenging as the system scales. Software faults are inevitable, and finding them is challenging.

Throughout history, there have been several attempts to solve challenges in software testing. The existing literature shows that search-based white-box fuzzing is one such approach, which has proven to be efficient [30, 7]. At the same time, to achieve fully automated system-level testing, handling external communications is one of the major challenges and time-consuming tasks. These days web applications heavily rely on external web services for several other purposes, such as payments (e.g., Stripe[1]) and authentication (e.g., OAuth[2]). For example, in one of our real-world case studies (i.e., *catwatch*) used in our experiments, it heavily relies on GitHub [3] Representational State Transfer (REST) Application Programming Interfaces (API)s to execute its goals. It is a common practice to use hand-crafted mock services or to use *Mock Objects* to simulate such external service dependencies [55, 10]. However, using *Mock Objects* and using a dynamic mock server are two different approaches. Several industry-

[1] `https://stripe.com/`
[2] `https://oauth.net`
[3] `https://github.com/`

standard tools and libraries facilitate the setup and operation of mock external web services with minimal effort, including Mockoon,[4] Postman,[5] and WireMock[6]. These tools share core functionality that enables the creation of configurable mock HTTP-based web services for testing. However, none of these tools provides the capability to automatically generate such mock web services. Writing every mock server by hand also requires significant effort. A complementary approach is to use record-and-replay to create such mock services. However, if the external web services are not owned by the same developer, it is not always feasible to create a mock server that covers all possible cases, especially when using the record-and-replay approach. Additionally, there is no guarantee that it is possible to create mock servers to test edge cases with less effort. Recognizing these challenges, our work primarily focused on creating such mock servers automatically. Building a configurable mock server from scratch is a massive engineering effort somewhat similar to building a software fuzzer. Due to this, we chose the widely used library, WireMock[6], from the available tools for Java Virtual Machine (JVM)-based applications as the base for the configurable mock server.

In the example shown in Figure 1.1, the developer of *catwatch* may or may not have a full understanding of the third-party services to create a robust mock service to test all the possible scenarios. To the best of our knowledge, there is a research gap for developing techniques to generate mock services in a fully automated manner without requiring any prior knowledge of such external service dependencies. In Article 1 and Article 2, we present our novel search-based, white-box fuzzing techniques to automatically generate mock external web services for the JVM-based applications as an academic proof.

Meanwhile, in pursuit of developing novel techniques to automatically generate mock services, we identified another unsolved problem required to achieve fully automated generation. In statically typed languages used in web applications (e.g., Java, Kotlin, and Go), it is common to use the Data Transfer Object (DTO) design pattern to represent requests and responses from other services. At the same time, REST APIs

[4] `https://mockoon.com/`
[5] `https://www.getpostman.com/`
[6] `https://wiremock.org/`

```java
public interface HttpConnector {
    /**
     * Opens a connection to the given URL.
     */
    HttpURLConnection connect(URL url) throws IOException;

    /**
     * Default implementation that uses {@link URL#openConnection()}.
     */
    HttpConnector DEFAULT = new ImpatientHttpConnector(new HttpConnector()
    {
        public HttpURLConnection connect(URL url) throws IOException {
            return (HttpURLConnection) url.openConnection();
        }
    });
}

```

Figure 1.1: The code snippet taken from one of our open-source case studies (e.g., *catwatch*) shows how an external web service connection is established inside the system under test.

currently use JavaScript Object Notation (JSON) as the de facto standard for data exchange, alongside Extensible Markup Language (XML) [49, 14]. Automatically generating such well-formed inputs (e.g., based on JSON schemas) for search-based white-box fuzzing is one of the several challenges in achieving fully automated system-level testing. Moreover, such complex input formats are not only used in DTOs but also to represent application configurations. For example, it is common practice to use *YAML Ain't Markup Language* (YAML) (also known as *Yet Another Markup Language*) configuration files, with Docker [7] and other popular applications. In the literature, there are works done to use user-specified grammar to deal with such complex inputs [29]. In *catwatch*, the application uses a software development kit (SDK) from *GitHub* to perform its goal. In this instance, if the complete documentation of the SDK is available, it is possible to create a user-specified grammar. However, if the application uses a proprietary SDK, creating such a system using user-specified grammars may not always be feasible. Moreover, to achieve fully automated testing, it is important to reduce the manual human intervention to near zero. In Article 3, we present our novel techniques

[7] https://www.docker.com/

to infer JSON-based schemas in a fully automated manner for search-based, white-box fuzzing.

However, while external communications not only help to achieve the goal of the application, they also introduce new threats to the application and make it vulnerable. For example, if the system under test (SUT) heavily relies on external communications to execute some goal, and such connections are not owned by the same developer, it could lead to some unfavourable outcomes. Particularly, supply chain compromise and trusted relationship attacks have become increasingly common [3]. A recent example is the NPM supply chain attack. Such external communication may not be from the same developer; instead, it is part of a dependency used in the application. In Article 4, we present our novel techniques to detect and exploit server-side request forgery (SSRF) automatically in white-box and black-box fuzzing.

Engineering a new software fuzzer from scratch with white-box and black-box capabilities in a huge effort. Due to this reason, we chose the open-source fuzzer, EvoMaster [12, 16, 13]. Based on the existing literature, it performs well in terms of coverage compared to other white-box fuzzers [36, 62]. Additionally, EvoMaster has software functionalities to support white-box and black-box testing for web applications [62], particularly REST APIs. We implemented our novel techniques as an extension to EvoMaster. As a result, we will generate self-contained test suites with dynamic mocking capabilities and SSRF detection.

To evaluate the efficiency of our novel techniques and to answer the research questions, we chose multiple open-source case studies and one industry case study. In each article, we used multiple case studies mainly from Web Fuzzing Dataset (WFD) [8], formerly known as EMB corpus [15]. Additionally, we developed multiple synthetic case studies with examples adapted from real-world case studies. We discussed in detail under each article about them. Moreover, in Article 4, we used two more open-source case studies for SSRF detection, which have Common Vulnerabilities and Exposures (CVE) reported.[9]

[8] https://github.com/WebFuzzing/Dataset/
[9] https://www.cve.org

The thesis contains two main parts. Part I is about summarizing the research work for the entire thesis, which is organized as follows. Firstly, Section 1.2 is the background that provides the necessary information for understanding the thesis. Following that, the research methodology is presented in Section 1.3. The Section 1.4 briefly discusses the technical details of automated mocking and vulnerability detection. Section 1.5 presents the evaluation method used to validate our novel techniques, followed by Section 1.6, which discusses the outcomes and findings from the articles. The Section 1.7 discusses the details about the contributions made by this research. Finally in Part I, Section 1.8 concludes the thesis summary. Part II contains the articles included in this thesis.

1.2 Background

This section outlines the background and motivation for the research conducted in this thesis. The section is organized into four primary areas. Firstly, Subsection 1.2.1 provides a brief about search-based software testing and its benefits. Secondly, Subsection 1.2.2 provides the background and motivation for developing novel techniques to automate mock generation. Thirdly, Subsection 1.2.3 discusses the details about automated schema inference for JSON schemas. Finally, Subsection 1.2.4 discusses the details about the automated SSRF detection.

1.2.1 Search-based Software Testing

Software testing is a fundamental discipline within the domain of software engineering that focuses on the systematic evaluation of software systems to ensure their accuracy, dependability, and performance. In general, software testing is performed under controlled conditions to identify defects, validate functionality, and assess compliance with specified requirements. Different testing methodologies, including black-box, white-box, and gray-box testing, are used to address different aspects of software behavior and uncover potential vulnerabilities. Additionally, the testing process is typically divided into several levels, such as unit testing, integration testing, system testing, and acceptance testing, each targeting specific objectives and contributing to the overall quality assurance framework [47]. Recent advancements in automated testing tools and techniques have significantly enhanced the efficiency and effectiveness of the testing process [27]. Software systems are becoming increasingly complex and integral to daily life. The importance of rigorous testing methodologies in minimizing risks and ensuring software reliability cannot be overstated [44, 63, 49]. Previous studies show that search-based algorithms inspired from nature, are very successful in solving software engineering problems (i.e., search-based software engineering) such as software testing (i.e., search-based software testing) [33, 5].

Fuzzing is a software testing technique that uses random or invalid inputs to be given to the software to analyze its behavior [30]. Fuzzing is one of the most effective

methods to detect software faults (in particular security-related faults) and to achieve higher code coverage [28, 41, 64]. Particularly, white-box and black-box fuzzing of REST-based web services has gained a major amount of attention among researchers lately [31, 36, 62].

White-box fuzzers have been shown to be highly effective in numerous cases [30, 43, 7]. In white-box fuzzing, internal details of the SUT, including source code, binaries, or bytecode, are accessed to inform the testing process. This information can be used to design heuristics to improve search and produce better test cases with higher code coverage and fault-detection capabilities [14].

1.2.2 Automated Mocking

When testing web services in isolation, external web service dependencies are typically mocked to avoid conflicts [55]. Mocking is a popular strategy in unit testing to handle dependencies [10]. *Mock Objects* is a common approach to isolate a dependency from its original ones [39, 10]. A mock object can be used to represent the response to a call made to an external service or database. Especially in tests, this can be programmatically configured without executing the original code.

Mock Objects help to expedite software testing and allow testing of the functionalities individually [55]. There are many industrial frameworks available to handle dependencies during testing. Some examples are Mockito [10], EasyMock [11], and JMock [38]. However, creating mocks is still manual in these frameworks, and none of them can generate *Mock Objects* automatically.

The code block shown in the Figure 1.1 is taken from the *GitHub* software development kit (SDK) used in one of our case studies (i.e., *catwatch*) as an example. The class `HttpConnector` is responsible for opening a connection to the given Uniform Resource Locator (URL) using `java.net.HttpURLConnection`. The connectivity to the external web service determines the program's further execution. Since the program depends on the response from the external web service to execute its goal, suc-

[10] `https://site.mockito.org/`
[11] `https://easymock.org/`

cessfully passing that line of code is vital.

In situations shown in Figure 1.1, mocking can be a vital resource to perform automated system-level testing. However, in system-level testing, *Mock Objects* cannot always be used directly if the SUT is executed via user interfaces (e.g., a GUI or network calls for external web services).

Mocking entire external web services is different from mocking the network and dependencies through *Mock Objects*, although they share the same goal. There are several tools and libraries available in the industry that can help to set up and run a mock external web service with less effort, such as Mockoon [4], Postman [5], and WireMock [6]. All of them share common functionalities to create configurable mock HTTP-based web services for testing. However, none of them have the functionality to automatically generate such mock web services. Additionally, creating handwritten mock external web services can be challenging and time-consuming [6]. Furthermore, the process becomes convoluted if third-party interactions are executed using a provided software development kit (SDK), since knowledge of the interactions may not be available unless the SDK is provided or reverse-engineered.

In this particular example shown in Figure 1.1, the *GitHub* SDK will try to connect to *GitHub* web APIs using *HttpURLConnection.openConnection()* to fetch information as in Line 12. In existing approaches, such a connection can be imitated using hand-crafted mock instances or *Mock Objects* of the *GitHub* APIs [55, 39, 10]. However, to perform fully automated system-level testing, finding a way to automatically create those mock instances could improve efficiency [6]. Moreover, if we can mock the specific external web service (e.g., the *GitHub* Web API), we can test the SUT for edge cases that may not be handled within the SUT.

To the best of our knowledge, no technique in the scientific literature can fully automate mock generation for handling external web service dependencies. The work done in [10, 34] is perhaps the closest in the literature to what we are trying to achieve. However, there are some significant differences. In a previous study [34], only artificial classes were evaluated; in the meantime, we show that our techniques can scale to real systems (including an industrial API) as presented in Articles 1 and 2. In addition,

previous studies that mock the networking interfaces of the Java standard library as an input space for search-based fuzzers have shown significant improvements in code coverage [11, 10]. This indicates that significant improvements in code coverage can be achieved by automating the mocking of external web services. In this thesis, we present search-based white-box fuzzing techniques that will automatically generate mock external web services.

1.2.3 Automated Schema Inference

Intertwined REST and JSON

REST APIs have gradually emerged as the primary means of communication among various services, encompassing cloud SaaS (Software as a Service) and especially in microservices [25, 49]. In the example shown in Figure 1.1, the *GitHub* provides multiple REST APIs, and they are encapsulated within an SDK.

REST is an architectural style that emphasizes stateless communication and uses standard HTTP methods such as GET, POST, PUT, PATCH, and DELETE to perform operations on resources. The simplicity, scalability, and flexible nature of REST APIs make them suitable for a wide range of applications. In general, REST APIs typically use either JSON or XML response schemas [7]. The OpenAPI specification provides a standardized interface for REST APIs, facilitating automated testing of a substantial number of REST APIs [2, 31, 7]. Besides the popularity of REST APIs in the industry, there have been previous studies conducted using REST APIs in academia as well [57, 36, 20], especially in the context of search-based software engineering [61, 42, 31].

Simultaneously, JavaScript Object Notation (JSON) is a lightweight data interchange format used in web services as an alternative to Extensible Markup Language (XML). JSON is a key-value pair format that allows for the representation of complex data structures, including arrays and nested objects [1]. JSON payloads are widely used in the industry, especially in REST-based web services [48, 54]. Furthermore, JSON is also a popular schema format used in academic research [14, 40, 58, 19].

Grammars In Fuzzing

Grammar-based fuzzing offers a viable approach to overcoming the challenges of generating JSON schemas for mock services. In grammar-based fuzzing, highly effective test inputs are generated based on user-specified grammar. With the help of formal grammar specifications, fuzzers can generate and manipulate well-formed input data. An attempt to combine search-based testing with grammar-based fuzzing showed significant improvement in branch coverage for JSON-related classes [52]. Previous studies indicate that the use of grammar-based fuzzing significantly improves code coverage and fault detection rate, especially with input formats such as JSON [23, 58]. Grammar specification typically can be user-specified or can be inferred from documentation (e.g., OpenAPI). Another approach is mining such grammars [32].

The existing literature focuses on grammar-based inputs for fuzzing the endpoints of the SUT. To enable effective, fully automated system-level testing, we need to automatically generate mock responses with valid syntax (e.g., *JSON* responses), regardless of whether any formal schema is present or not (e.g., the work in [34] requires specifying XML Schema Definition (XSD) files manually with the schema of the response messages). Therefore, we need to identify valid JSON response schemas without manual intervention. It is possible to infer such a schema from the documentation, if available. For example, in the example shown in Figure 1.1, it is possible to obtain the requests and responses from the *GitHub* documentation to create mock instances. However, this may not always be feasible. For instance, if the SUT uses an SDK from a third-party vendor, it may not always be possible to determine the input format to create a grammar for further use.

To overcome this challenge, we took an entirely different approach using instrumentation for schema extraction rather than relying on grammar-based techniques (e.g., grammar mining or user-provided grammar). To the best of our knowledge, there exists no prior research on automated multi-level schema extraction using white-box techniques. We developed scalable techniques using bytecode instrumentation to automate schema extraction and generate useful mock external web services, as pre-

sented in Article 3. Especially, our techniques can extract schemas with complex data structures, such as objects and arrays.

1.2.4 Automated SSRF Detection

Software security is an integral component of any software system. There are a significant number of proposals available from academia about how to build a secure software system [24], and there are numerous tools and methods available to detect vulnerabilities in advance [37]. However, to the best of our knowledge, there is still a gap between understanding these phenomena and providing effective results [31, 4, 44, 6]. Existing works from various researchers have focused mostly on cross-site scripting (XSS) [17], SQL injection (SQLi) [56, 18], and XML injection [35] using search-based techniques. Although there is a research gap in detecting software vulnerabilities (e.g., SSRF) using search-based algorithms [4]. Typically, SSRF occurs when the application is tricked into making a call to an unintended external service/location, which could lead to unauthorized information disclosure, remote code execution, and more. Furthermore, SSRF is listed as one of the top ten in OWASP 2021 [53].

To give a perspective, as shown in Figure 1.1, hypothetically, in case if the *GitHub* API is compromised and has a malicious value in the response, it may affect the SUT in several ways. Typically, this can be prevented by input sanitization. However, based on our observation from our case studies, it is uncommon to implement such input sanitization when it comes to connection with external web services. In such a case, if the SUT uses it without proper sanitization in a critical component (i.e., system command execution), there is a possibility this may result in potential security incidents, such as scenarios commonly known as supply chain attacks and dependency chain abuse [51].

To elaborate more on the claim, Figure 1.2 contains a code snippet taken from one of our real-world case studies used in Article 4. In our previous example shown in Figure 1.1, communication to the external API is defined. However, in Figure 1.2, the application takes the input from the user or the client application to connect to a remote

```
1 // If the component parameter is specified, this function returns a string
    (or int in case of PHP_URL_PORT)
2 /** @var string $path */
3 $path = parse_url($url, PHP_URL_PATH);
4 $extension = '.' . pathinfo($path, PATHINFO_EXTENSION);
5
6 if ($extension !== '.') {
7     // Validate photo extension even when `$create->add()` will do later.
8     // This prevents us from downloading unsupported files.
9     BaseMediaFile::assertIsSupportedOrAcceptedFileExtension($extension);
10 }
11
12 // Download file
13 $downloaded_file = new DownloadedFile($url);
14
```

Figure 1.2: Code snippet responsible for downloading images from a remote location, taken from one of our real-world case studies, *Lychee*.

location to download images. In Article 4, we discussed in detail our observation that this can be exploited further to gain more access to the server, although a validation exists in Line 9 in Figure 1.2. In addition, in another real-world case study used in the article, we observed that the application (i.e., *microcks*) downloads configuration from remote server without proper checks on the file.

In the existing literature, there are several methods have been proposed for automated vulnerability discovery [22, 59, 21, 35, 60]. For example, a previous study [59] proposes methods to automatically detect SSRF only in PHP web applications using dynamic tainting through a graphical user interface inputs. Apart from that, to the best of our knowledge, in the existing literature there are only two existing approaches similar to our work presented in the Article 4 in terms of detecting SSRF for REST APIs [60, 22]. However, in the study [60], no evidence or details are provided on how the techniques work for detecting SSRF, nor is it clear where the approach is automated. Furthermore, since the prototype is not available (referenced link[12] on GitHub seems broken), it is impossible to verify the approach without investigating the source code. In another study related to detecting security vulnerabilities in REST APIs [22], presents an approach that can detect SSRF. However, the presented approach requires

[12] https://github.com/apif-tool/APIF_tool_2024

a human-in-the-loop to review the prepared documentation, due to this fact it is not fully automated.

Overall, existing work in the literature falls short in terms of a fully automated, language-agnostic approach for SSRF detection in REST APIs. Our work, presented in Article 4, provides fully automated SSRF detection techniques in both white-box and black-box fuzzing. Furthermore, in our approach, we generated self-contained test suites that can be used to reproduce and verify the existence of SSRF in REST APIs.

1.3 Research Methodology

This section presents the research method we used throughout the thesis. This research is part of an ongoing project funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (EAST project, grant agreement No. 864972) that aims to improve understanding of the intrinsic characteristics of web/enterprise systems related to their security. We took an empirical software engineering approach to conduct the research. Initially, we designed and developed our novel techniques, then conducted experiments to produce evidence and analyzed the results using the statistical methods recommended in the software engineering literature [8, 9].

1.3.1 Research Questions

At each step of the research, we aim to answer three sets of questions for each challenge we seek to solve. The first step is to assess the effectiveness of the techniques presented in Article 1 using the following research questions.

RQ1: What is the impact on code coverage and fault-finding of our novel search-based approach for external service mocking?

Consequently, building on the work presented in Article 1, we expanded our work to answer the following research questions to validate our techniques.

RQ2: Is our novel approach capable of automatically generating mock external web services for artificial, hand-crafted examples?

RQ3: Compared to the performance of existing white-box fuzzing techniques, how much improvement can our mock external web service generation achieve in terms of code coverage and fault detection on real-world applications?

RQ4: When external services are inaccessible, how effective is our approach in generating mock external web services?

To solve the challenges identified in Article 2 related to automated schema inference of JSON documents, we expanded our study to answer the following research question.

RQ5: Are our novel techniques capable of automatically inferring JSON-based schemas for case studies adapted from real-world complex examples?

In the end, we aim to answer the following research questions related to automatically detecting SSRF presented in Article 4.

RQ6: Can our novel techniques automatically detect SSRF security faults in synthetic APIs?

RQ7: Can our novel techniques automatically detect SSRF security faults in reported vulnerabilities from real-world APIs?

1.3.2 Implementations

As part of the research, to answer the research questions in Subsection 1.3.1, we developed our novel techniques in Java and Kotlin as an extension to EvoMaster.

Case Studies

Initially, we developed various end-to-end tests in EvoMaster to represent the outcomes we aim to achieve. For example, to address **RQ1**, we developed synthetic case studies based on real-world use cases. The difference between a synthetic case study and a real-world case study is that, in a synthetic case study, we developed a few endpoints and focused only on the key business logic. On the contrary, a real-world case study contains more endpoints with complex business logics. For various purposes, we developed a range of synthetic case studies representing various real-world scenarios to aid our development. Later, those case studies were used in our experiments as academic proof of concept to demonstrate the effectiveness of our novel techniques. Primarily, we relied on open-source case studies from WFD [8]. Additionally, we collected a few more real-world case studies and contributed to WFD (i.e., *pay-publicapi*

is a new case study added to the dataset). Furthermore, we used two more real-world case studies to test our automated SSRF detection techniques in Article 4.

Extensions to EvoMaster

Firstly, to implement our techniques, we extended EvoMaster's instrumentation capabilities and collected additional heuristics based on the search's focus. Such instrumentation allows us to collect information about external service calls, JSON schemas, and more. Throughout the search, we ensure that method replacements preserve the semantics of the original methods without interfering with the execution of the SUT. Additionally, we introduced new configurable options to customize the search based on the needs of our experiments.

Furthermore, we extended EvoMaster fitness function to support our fitness criteria depending on the use case. Finally, we modified the test suite writer for EvoMaster to generate self-contained tests with mock servers and other logics.

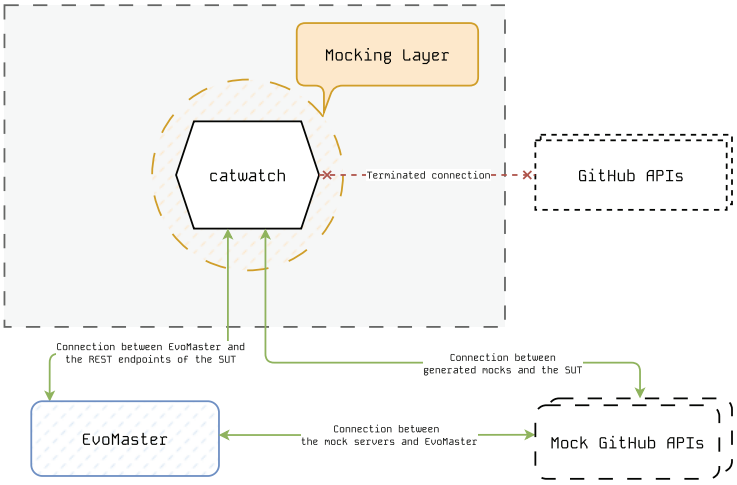


Figure 1.3: The diagram illustrates the dependencies of the selected API, *catwatch*, on external web services from *GitHub*. The *Mocking Layer* serves as a virtual boundary that terminates interactions with actual external web services.

1.4 Automated Mocking and Vulnerability Detection

This section provides an overview of the interconnections among all the articles in this thesis, offering a comprehensive perspective. The section is divided into three main subsections as follows. Subsection 1.4.1 provides an overview of automated mocking presented in Articles 1 and 2. Secondly, Subsection 1.4.2 discusses the higher-level details about automated multi-phase schema inference presented in Article 3. Finally, Subsection 1.4.3 highlights the automated SSRF detection presented in Article 4.

1.4.1 Automated Mocking

Figure 1.3 illustrates the higher-level architecture underlying the implementation of the mocking techniques described in Articles 1 and 2. In general, when an application makes a call within a JVM-based application, it tries to resolve the host using `InetAddress`. Once the host is resolved, the application proceeds to the next step and initiates a socket connection on the specified port; otherwise, it throws an exception. When a

```
1 wireMock__http__json_test__10877 = WireMockServer(WireMockConfiguration()
2     .bindAddress("127.0.0.26")
3     .port(10877)
4     .extensions(ResponseTemplateTransformer(false)))
5 wireMock__http__json_test__10877.start()
6
```

Figure 1.4: Mock server initialization in the generated test suite.

secure connection (e.g., Secure Sockets Layer (SSL)) is required, the application initiates certificate validation on the client side using the PKIX (Public Key Infrastructure X.509) algorithm. Upon successful validation, the application proceeds with execution to initiate data transfer. Throughout this process, we extended EvoMaster's instrumentation capabilities to instrument the necessary classes to collect information such as protocol, hostname, port, HTTP method, and more. Simultaneously, we leverage this instrumentation to alter connection parameters, such as the hostname, so the application connects to our mock server instead of the actual service. Such instrumentation also helped us to collect more information about the HTTP request made by the SUT further down in the execution. Once we have enough information about the external service, we spin off the mock server for each host using WireMock[6] with a default response of HTTP 404 for all requests.

After the SUT successfully makes the first request to the mock server, we collect additional information about each request and begin configuring the mock servers with stubs to serve. Eventually, throughout the search, we randomly mutate the responses to test for various scenarios. During which, we used the techniques presented in Article 3, to infer the JSON schema automatically to produce more meaningful responses from the mock servers.

At the beginning of the test suite, we initiate the mock servers and terminate them at the end. Meanwhile, before each test, we reset and configure it with the relevant information needed for successful execution. Figure 1.4 showcases the mock server setup in the generated test suites under `@BeforeAll` of *JUnit* tests from one of our synthetic case studies. In Figure 1.4 Line 2, the mock server is initiated in a local IP address and in the port specified in the SUT in Line 3.

```
1 DnsCacheManipulator.setDnsCache("json.test", "127.0.0.26")
2
```

Figure 1.5: The code snippet responsible for DNS cache manipulation in the generated tests related to automated mocking.

In the generated test, we do not have the luxury of instrumentation. Due to this, we relied on Java reflection to manipulate the JVM's Domain Name System (DNS) cache using the `dns-cache-manipulator` from Alibaba to redirect the traffic to our mock servers.[13] Figure 1.5 shows the line of code that is responsible for configuring the DNS cache. We provided more details about the whole process in Article 2.

1.4.2 Automated Schema Inference

This subsection provides a higher-level picture of automated schema inference presented in Article 3. Once the connection is initiated successfully to our mock server, on the REST call, with our instrumentation and by modifying the taint analysis capabilities in EvoMaster, we were able to observe and collect schema information expected from the remote server using the Data Transfer Objects (DTO). As shown in Figure 1.3, when an application tries to parse the response from the *GitHub* APIs, the responses are represented using various DTOs. When an input string is parsed inside the application using JSON marshalling libraries (e.g., Jackson[14] or Gson[15]) to unmarshall it to JSON element, with the help of instrumentation, we managed to infer the schema from the parsed DTO.

Apart from the required responses from the SUT for a successful response, throughout the search, we mutated responses in mock servers to test for other scenarios based on HTTP response status codes, various request body payloads, and more. Figure 1.6 contains a generated test case for one of our artificial case studies demonstrating the mock server response setup with an *Array* of *Map* as elements. In Figure 1.6 Line 17, the response required for an HTTP 200 response by the application is configured to the

[13] https://github.com/alibaba/java-dns-cache-manipulator
[14] https://github.com/FasterXML/jackson
[15] https://github.com/google/gson

```
/**
* Calls:
* (200) GET:/api/wm/jsonarray
*/
@Test @Timeout(60)
fun test_5_getOnJsonarrayReturnsContentUsingWireMock()  {
    DnsCacheManipulator.setDnsCache("json.test", "127.0.0.26")
    assertNotNull(wireMock__http__json_test__10877)
    wireMock__http__json_test__10877.stubFor(
        get(urlEqualTo("/api/foo"))
        .atPriority(1)
        .willReturn(
            aResponse()
                .withHeader("Connection","close")
                .withHeader("Content-Type","application/json")
                .withStatus(201)
                .withBody("[" +
                    "{" +
                    "\"x\": 485," +
                    "\"cycle\": {" +
                    "\"y\": 506" +
                    "}" +
                    "}," +
                    "{" +
                    "\"cycle\": {" +
                    "\"y\": 690" +
                    "}" +
                    "}," +
                    "{" +
                    "\"x\": 711," +
                    "\"cycle\": {" +
                    "\"y\": -207" +
                    "}" +
                    "}" +
                    "]"
                )
            )
    )
    given().accept("*/*")
            .header("x-EMextraHeader123", "")
            .get("${baseUrlOfSut}/api/wm/jsonarray")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("text/plain")
            .body(containsString("OK X and Y"))

    wireMock__http__json_test__10877.resetAll()
}

```

Figure 1.6: Generated test to illustrate an automated multi-phase schema inference.

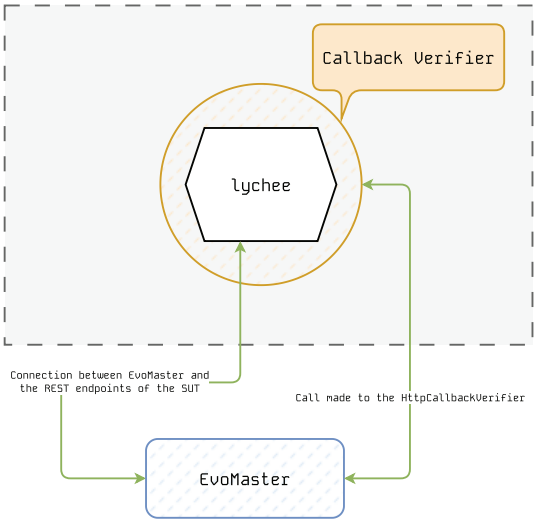


Figure 1.7: Illustration of a higher-level overview of the SSRF detection process.

mock server in Line 43. Further down the execution, test assert the response body in Line 46 in Figure 1.6.

At the end of the search process, test suites with self-contained mock servers for each external web service will be generated.

1.4.3 Automated SSRF Detection

Building on the foundational idea presented in Articles 1 and 2 about automated mocking, we took a similar approach to create the `SSRFAnalyser` with a `HttpCallbackVerifier` presented in Article 4. In automated mocking discussed in Subsection 1.4.1, we relied on the instrumentation to identify the calls made. In all cases of SSRF, typically the external service information provided by the client application or by the user as an input. Due to this, it was not necessary to instrument the classes related to the network connection as in Subsection 1.4.1. Since there is no requirement for instrumentation, we used a regex-based deterministic approach to identify the input parameters in the HTTP request which are likely to be a URL. Instead of terminating

```
1 // Fault202. Server-Side Request Forgery (SSRF). sensorUrl.
2 given().accept("*/*")
3     .header("x-EMextraHeader123", "")
4     .contentType("application/json")
5     .body(" { " +
6         " \"sensorUrl\": \"http://localhost:50249/EM_SSRF_0\" " +
7         " } ")
8     .post("${baseUrlOfSut}/api/fetch")
9     .then()
10    .statusCode(200)
11    .assertThat()
12    .contentType("text/plain")
13    .body(containsString("OK"))
14
```

Figure 1.8: The code from the generated test for SSRF detection is responsible for making the REST call to the SUT.

the external service connection as shown in Figure 1.3, we provided the URL of our malicious mock server initiated inside `HttpCallbackVerifier` as the destination in the identified input parameters, as shown in Figure 1.7. This allowed to extend our novel techniques to work both in white-box and black-box fuzzing.

Once the call was successfully made to our malicious mock server, we marked the respective individual (e.g., *RESTIndividual*) contains the REST call as exploitable with additional information about the fault category as indicated in Figure 1.8 Line 1. During test suite generation, similar to the automated mocking presented in Subsection 1.4.1, we generate a malicious mock server that acts as the `HttCallbackVerifier` to assert the successful exploitation of SSRF.

Under each test, we configure the verifier to serve the malicious payload as shown in Line 2 in Figure 1.9.

Before executing the actual REST call in the SUT, we make sure there are no calls made to the verifier, as shown in Figure 1.10.

Figure 1.11 shows a similar assertion made after the REST call is executed to ensure the successful exploitation of the vulnerability.

```
httpCallbackVerifier50249.resetAll()
httpCallbackVerifier50249.stubFor(
    get("/EM_SSRF_0")
        .withMetadata(Metadata.metadata().attr("ssrf", "POST:/api/fetch"))
        .atPriority(1)
        .willReturn(
            aResponse()
                .withStatus(200)
                .withBody("SSRF")
        )
)

```

Figure 1.9: Code to configure the `HttpCallbackVerifier` in the generated test suites.

```
// Verifying that there are no requests made to HttpCallbackVerifier before
    test execution.
assertFalse(httpCallbackVerifier50249
    .allServeEvents
    .filter { it.wasMatched && it.stubMapping.metadata != null }
    .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/fetch" }
)

```

Figure 1.10: The code responsible for asserting no calls made to the `HttpCallback-Verifier` before the REST call is executed.

```
        // Verifying that the request is successfully made to
    HttpCallbackVerifier after test execution.
        assertTrue(httpCallbackVerifier50249
            .allServeEvents
            .filter { it.wasMatched && it.stubMapping.metadata != null }
            .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/
    fetch" }
        )

```

Figure 1.11: The code responsible for asserting no calls made to the `HttpCallback-Verifier` after the REST call is executed.

1.5 Evaluation

This section presents the evaluation process carried out in all the articles and the results. First Subsection 1.5.1, discuss the case studies used in the experiments and the selection criteria. Secondly, Subsection 1.5.2, empirical analysis of the experiments results.

1.5.1 Case Studies

Real-world Case Studies

We chose our real-world open-source case studies to conduct our empirical study from Web Fuzzing Dataset [8], formerly known as EMB [15] and one industrial case study (i.e., *ind1*) in Articles 1 and 2. WFD is an open-source corpus composed of open-source web/enterprise APIs for scientific research and previously used in other studies [15]. In addition, we added one more case study (i.e., *pay-publicapi*) to the dataset which we used in Article 2.

The case studies selected from the WFD dataset are as follows:

catwatch is a web application which allows fetching *GitHub* statistics for *GitHub* accounts, processes, and saves the data in a database and provides them via a RESTful API. The application heavily relies on *GitHub* APIs to perform its core functions.

cwa-verification is a component of the official Corona-Warn-App for Germany. This case study is a part of a microservice-based architecture which contains various other services, including mobile and web apps. The tested backend server relies on other services in the microservice architecture to complete its tasks.

genome-nexus is an interpretation tool which aggregates information from various other online services about genetic variants in cancer.

pay-publicapi is an open-source API from the government of the United Kingdom for government or public sector organizations that want to take payments. The API provides the ability to take payments, issue refunds, and run reports on all payments. This API is currently being used by partners from across the public sector, including

the NHS, MOJ, police forces, and local authorities.

ind1 is a component in an e-commerce system. It deals with authentication using an external Auth0 server, and it processes monetary transactions using Stripe [1].

Article 4 presents two more case studies involving real-world reported CVE database [9].

microcks is a platform for turning your API and microservices assets - *OpenAPI specs*, *AsyncAPI specs*, *gRPC protobuf*, *GraphQL schema*, *Postman collections*, *SoapUI projects* - into live mocks in seconds.[16]

lychee is a free, open-source photo-management PHP tool that runs on your server or webspace.[17]

Synthetic Case Studies

In addition to the real-world case studies, we developed several synthetic case studies to simulate real-world examples for experimental purposes. Such case studies were used to aid the development of our techniques and to run experiments. In Article 3, we developed case studies with various business logic adapted from real-world open-source applications, as outlined in WFD [8]. Similarly, in Article 4, as an academic proof-of-concept, we developed three synthetic case studies that present various SSRF scenarios.

1.5.2 Empirical Analysis

In Articles 1 and 2, our goal is to analyze the impacts our techniques can have in terms of code coverage and fault detection. To answer the research questions, our experiments measure multiple metrics, including code coverage and fault-finding rate. To account for the randomness of search-based algorithms when analyzing their results, we employed common statistical tests recommended in the literature [8, 9] as mentioned in Section 1.3. In particular, we employed a Mann–Whitney U test (i.e., p-value) and Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) to perform comparison analysis between de-

[16] `https://microcks.io`
[17] `https://lycheeorg.dev`

fault settings (i.e., *Base*) of EvoMaster and against our approach (i.e., *Mocking*) in terms of code coverage and fault detection.

With the Mann-Whitney U test, if the p-value is less than the significance level (i.e., 0.05), it indicates that the two compared groups (i.e., *Base* and *Mocking*) have statistically significant differences. Otherwise, there is insufficient evidence to support the claim that the difference is significant. Comparing *Mocking* with *Base*, the Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) measures how likely *Mocking* can perform better than *Base*. $\hat{A}_{12} = 0.5$ represents no effect. If it $\hat{A}_{12}$ surpasses 0.5, it indicates that *Mocking* has more chances to achieve better results than *Base*.

In Articles 3 and 4, our goal is to produce evidence (i.e., usable generated tests) that our techniques work. We utilized synthetic case studies that simulate real-world scenarios to automatically generate tests, providing evidence that the techniques are effective. Additionally, in Article 4, we evaluated our techniques against two real-world case studies which have CVEs reported. In our experiments, we successfully produced tests with all the necessary assertions to prove that our techniques work.

1.6 Discussion

In this section, we discuss the results presented in each article.

Overall, based on our experiments, our novel techniques perform well across both artificial and real-world APIs.

In Articles 1 and 2, our novel techniques performed well in all the synthetic case studies. In the meantime, such results cannot be achieved with the existing black-box approach [31]. Therefore, our novel techniques advance the state of the art in fuzzing Web APIs.

However, in both Articles 1 and 2, outcomes from the synthetic APIs do not imply that our novel techniques provide useful improvements in real-world cases. Further experiments with real-world case studies in both articles support our claim that our novel techniques also provide significant improvements in real-world APIs. This can be explained by the fact that, with mocking, we can efficiently test several different error scenarios.

Not all APIs are the same. Handling external services is only one of the current main open problems in fuzzing REST APIs (as identified in a study [62]). Articles 1 and 2 addressed one of the major challenges identified in a study [62], by providing an automated working solution, implemented in an open-source, state-of-the-art fuzzer (i.e., EvoMaster). Still, several research challenges need to be addressed to further improve the results.

In white-box fuzzing, heuristic collection requires instrumentation in the code, which increases computational cost. As a result of attempts to handle the mock servers, they add to the computational overhead, resulting in fewer tests that can be evaluated within the same time budget (e.g., 1-hour fuzzing sessions). Such could lead to exploring a smaller subset of the search space, leading to worse outcomes. Whether the trade-off between the quality of a fitness function and its computational cost pays off is not guaranteed. The novel techniques presented in both articles comprise many components that could affect the computational cost of each fitness evaluation. Through an indirect measure of performance, cost can be assessed by the number of HTTP calls

made during the session within the given time, indicating that the higher the computational cost, the fewer calls are made. For example, in our real-world cases such as *catwatch*, *genome-nexus*, and *pay-publicapi*, we observed a significant drop in the number of HTTP calls within the search budget. However, this cannot be directly linked to overhead with the heuristic, since our fitness function cost is not constant when running mock servers. For instance, requests with HTTP 4xx responses take less time than those with HTTP 2xx responses, which execute more code. In our empirical studies, we observed that the trade-off benefited from fewer calls, higher code coverage, and improved fault detection. Although when the SUT scales up with a large amount of external calls, whether it is possible to achieve better results requires further empirical investigations.

Apart from the computational cost, we observed that not having valid inputs for complex data structures in JSON documents results in lower performance during the experiments presented in Articles 1 and 2. As a solution, we presented our novel techniques of automated multiphase complex JSON schemas (i.e., lists, arrays, and sets) inference in Article 3. Our novel techniques are lightweight and scalable, as it does not require any grammar specification or complex static code analysis [23]. As the application grows, our techniques remain effective as they rely on actual input values tracked dynamically through test execution. However, our novel techniques fall short in terms of handling tree-based data structures (e.g., `JsonNodes`) to represent JSON documents in libraries like Jackson[14].

Building on the foundational ideas of automated mocking presented in Articles 1 and 2, we explored novel ways to perform fully automated vulnerability detection involving such external service communications, such as SSRF. Typically, in previous examples, the application does not take the external services communication information from the user through REST calls unless there is a necessity. However, in some cases, the application may use a user-provided URL for purposes such as fetching remote data from a sensor in an IoT platform or downloading a file (e.g., images or configuration). In such instances, with improper input sanitization, this could lead to minor or catastrophic inconveniences.

In Article 4, we presented our novel techniques to automatically detect SSRF in applications and provide a proof of exploitation in a self-contained test suite with malicious mock servers. The empirical results show that our techniques can detect and exploit most types of SSRF. However, our techniques fall short of automatically detecting second-order SSRF, as it requires chained REST calls. Furthermore, our novel techniques are language-agnostic, as demonstrated in the article with real-world case studies written in various languages (e.g., Java and PHP); this can be adapted to any other fuzzers. With a configurable `HttpCallbackVerifier`, instead of using a static payload, we can expand the techniques to look for other cases where SSRF could lead to other injection-based vulnerabilities (e.g., SQL injection). For example, in reported CVE (CVE-2024-35451) [50] for *LinkStack* [1] leads to a Command Injection [45] through SSRF.

1.7 Contributions

This section outlines the key contributions of the research.

A summary of primary contributions from the work presented in the articles of this thesis,

- Collection of search-based novel techniques to automatically generate HTTP-based mock servers to support system-level testing for REST APIs as presented in Articles 1 and 2.
- Novel search-based techniques were introduced to automatically infer JSON-based schemas, as detailed in Article 3.
- Novel search-based techniques were proposed to detect and exploit server-side request forgery (SSRF) vulnerabilities in REST APIs, as presented in Article 4.
- Significant extensions were made to the existing open-source search-based white-box fuzzer EvoMaster.
- Empirical studies were conducted on both open-source and industrial APIs, providing evidence that these techniques address the challenges in automated software testing.

As this research work is part of an ongoing project, the articles that are not included in the thesis provide significant insights into the steps taken throughout the journey. Furthermore, the article published in the Doctoral Symposium of **2024 IEEE Conference on Software Testing, Verification and Validation (ICST)** was excluded from the thesis due to the evolution of the research from the original plan described in the Doctoral Symposium article.

Additionally, as part of building case studies, we developed an open-source application intended to cover all possible vulnerabilities in web applications, named *Clerk*[18]. *Clerk* is used mainly in the work presented in Article 4 to test our techniques in black-box fuzzing.

[18] https://github.com/seran/clerk

1.8 Conclusion and Future Work

In this thesis, we presented a collection of articles discussing novel techniques for automatically generating mock external services using search-based methods and automatically detecting SSRF using a similar mocking approach for REST APIs.

The work presented in Articles 1 and 2 and the work presented in Article 3, to the best of our knowledge, is the first work in the literature that provides a working solution for this important problem. In the meantime, we identified several avenues for further enhancements of our techniques as future work. For example, in automated mocking, current work focuses only on HTTP-based RESTful API. However, it is common to use other protocols, such as message queues (e.g., RabbitMQ[19]) and stream processors (e.g., Apache Kafka[20]), for external communication, especially in microservices [49]. Due to the asynchronous communication pattern, further investigation is required to extend our current work to achieve fully automated system-level testing.

In our work on automated schema inference presented in Article 3, we fall short in supporting other data structures for representing JSON documents. For example, libraries like Jackson[14] use tree-based data structures (e.g., `JsonNodes`) to represent JSON documents. We observed that such tree-based data structures appear in a few other real-world case studies in WFD [8]. Additionally, our focus is only on JSON-based schema. However, XML-based schema is another popular data interchange format used in APIs. Further investigation is required to support such cases to make it a complete solution.

The work presented in Article 4 mainly focuses on detecting and exploiting SSRF in REST APIs in most of the SSRF types except second-order SSRF. Automatically detecting second-order SSRF requires further investigation into how to identify and related chain methods, or, in some cases, SSRF might occur through background jobs. The complexity demands more extensive research in solving this challenge. Further, there is a research gap for developing novel techniques to automatically detect chained vulnerabilities via SSRF, as we discussed in Section 1.6. Additionally, based on the idea

[19] `https://www.rabbitmq.com/`
[20] `https://kafka.apache.org/`

of dynamic mocking using *HttpCallbackVerifier* like approach, we can detect other vulnerabilities from the external service communication entry points. Most existing work in the literature on automated vulnerability detection primarily focuses on detection from the primary endpoint of an SUT, such as REST APIs. By creating methods to detect vulnerabilities in other endpoints, such as external service communications, we can identify potential scenarios, such as supply chain attacks [26]. Such advancements will pave the way for fully automated vulnerability detection.

Bibliography

[1] Json schema specification wright draft 00. `https://datatracker.ietf.org/doc/html/draft-wright-json-schema-00`.

[2] Open api specification. `https://swagger.io/specification/`.

[3] Supply chain compromise, 2018.

[4] AHSAN, F., AND ANWER, F. A critical review on search-based security testing of programs. *Computational Intelligence: Select Proceedings of InCITe 2022* (2023), 207–225.

[5] ALI, S., BRIAND, L. C., HEMMATI, H., AND PANESAR-WALAWEGE, R. K. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering 36*, 6 (2009), 742–762.

[6] ARCURI, A. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering 23*, 4 (2018), 1959–1981.

[7] ARCURI, A. Automated black-and white-box testing of restful apis with evomaster. *IEEE Software 38*, 3 (2020), 72–78.

[8] ARCURI, A., AND BRIAND, L. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)* (2011), pp. 1–10.

[9] ARCURI, A., AND BRIAND, L. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR) 24*, 3 (2014), 219–250.

[10] ARCURI, A., FRASER, G., AND GALEOTTI, J. P. Generating tcp/udp network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 155–165.

[11] ARCURI, A., FRASER, G., AND JUST, R. Private api access and functional mocking in automated unit test generation. In *2017 IEEE international conference on software testing, verification and validation (ICST)* (2017), IEEE, pp. 126–137.

[12] ARCURI, A., GALEOTTI, J. P., MARCULESCU, B., AND ZHANG, M. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software 6*, 57 (2021), 2153.

[13] ARCURI, A., ZHANG, M., BELHADI, A., MARCULESCU, B., GOLMOHAMMADI, A., GALEOTTI, J. P., AND SERAN, S. Building an open-source system test generation tool: lessons learned and empirical analyses with evomaster. *Software Quality Journal* (2023), 1–44.

[14] ARCURI, A., ZHANG, M., AND GALEOTTI, J. P. Advanced white-box heuristics for search-based fuzzing of rest apis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2024).

[15] ARCURI, A., ZHANG, M., GOLMOHAMMADI, A., BELHADI, A., GALEOTTI, J. P., MARCULESCU, B., AND SERAN, S. EMB: A curated corpus of web/enterprise applications and library support for software testing research. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (2023), IEEE, pp. 433–442.

[16] ARCURI, A., ZHANG, M., SERAN, S., GALEOTTI, J. P., GOLMOHAMMADI, A., DUMAN, O., ALDASORO, A., AND GHIANNI, H. Tool report: Evomaster—black and white box search-based fuzzing for rest, graphql and rpc apis. *Automated Software Engineering 32*, 1 (2025), 1–11.

[17] AVANCINI, A., AND CECCATO, M. Towards security testing with taint analysis and genetic algorithms. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems* (2010), pp. 65–71.

[18] AZIZ, B., BADER, M., AND HIPPOLYTE, C. Search-based sql injection attacks testing using genetic programming. In *Genetic Programming: 19th European Conference, EuroGP 2016, Porto, Portugal, March 30-April 1, 2016, Proceedings 19* (2016), Springer, pp. 183–198.

[19] BALLESTEROS, I., DE BARRIO, L. E. B., FREDLUND, L.-A., AND MARINO, J. Tool demonstration: Testing json web services using jsongen. *biblioteca.sistedes.es* (2018).

[20] BANIAȘ, O., FLOREA, D., GYALAI, R., AND CURIAC, D.-I. Automated specification-based testing of rest apis. *Sensors 21*, 16 (2021), 5375.

[21] DENG, G., ZHANG, Z., LI, Y., LIU, Y., ZHANG, T., LIU, Y., YU, G., AND WANG, D. {NAUTILUS}: Automated {RESTful}{API} vulnerability detection. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 5593–5609.

[22] DU, W., LI, J., WANG, Y., CHEN, L., ZHAO, R., ZHU, J., HAN, Z., WANG, Y., AND XUE, Z. Vulnerability-oriented testing for restful apis. In *33rd USENIX Security Symposium (USENIX Security 24)* (2024), USENIX Association, pp. 739–755.

[23] EBERLEIN, M., NOLLER, Y., VOGEL, T., AND GRUNSKE, L. Evolutionary grammar-based fuzzing. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12* (2020), Springer, pp. 105–120.

[24] FELDERER, M., BÜCHLER, M., JOHNS, M., BRUCKER, A. D., BREU, R., AND PRETSCHNER, A. Security testing: A survey. In *Advances in Computers*, vol. 101. Elsevier, 2016, pp. 1–51.

[25] FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[26] FOUNDATION, T. O. M2: Inadequate supply chain security.

[27] GIAMATTEI, L., GUERRIERO, A., PIETRANTUONO, R., AND RUSSO, S. Automated functional and robustness testing of microservice architectures. *Journal of Systems and Software* (2023), 111857.

[28] GODEFROID, P. Fuzzing: Hack, art, and science. *Communications of the ACM 63*, 2 (2020), 70–76.

[29] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation* (2008), pp. 206–215.

[30] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: whitebox fuzzing for security testing. *Communications of the ACM 55*, 3 (2012), 40–44.

[31] GOLMOHAMMADI, A., ZHANG, M., AND ARCURI, A. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* (aug 2023).

[32] GOPINATH, R., MATHIS, B., AND ZELLER, A. Mining input grammars from dynamic control flow. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (2020), pp. 172–183.

[33] HARMAN, M., MANSOURI, S. A., AND ZHANG, Y. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR) 45*, 1 (2012), 11.

[34] HAVRIKOV, N., GAMBI, A., ZELLER, A., ARCURI, A., AND GALEOTTI, J. P. Generating unit tests with structured system interactions. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)* (2017), IEEE, pp. 30–33.

[35] JAN, S., PANICHELLA, A., ARCURI, A., AND BRIAND, L. Search-based multi-vulnerability testing of xml injections in web applications. *Empirical Software Engineering 24* (2019), 3696–3729.

[36] KIM, M., XIN, Q., SINHA, S., AND ORSO, A. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2022), ISSTA 2022, Association for Computing Machinery, p. 289–301.

[37] LESZCZYNA, R. Review of cybersecurity assessment methods: Applicability perspective. *Computers & Security 108* (2021), 102376.

[38] LIN, D., LIPING, F., JIAJIA, H., QINGZHAO, T., CHANGHUA, S., AND XIAOHUI, Z. Research on microservice application testing based on mock technology. In *2020 International Conference on Virtual Reality and Intelligent Systems (ICVRIS)* (2020), IEEE, pp. 815–819.

[39] MACKINNON, T., FREEMAN, S., AND CRAIG, P. Endo-testing: unit testing with mock objects. *Extreme programming examined* (2000), 287–301.

[40] MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)* (2012), IEEE, pp. 177–182.

[41] MANÈS, V. J., HAN, H., HAN, C., CHA, S. K., EGELE, M., SCHWARTZ, E. J., AND WOO, M. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering 47*, 11 (2019), 2312–2331.

[42] MARCULESCU, B., ZHANG, M., AND ARCURI, A. On the faults found in rest apis by automated test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 3 (2022), 1–43.

[43] MARTIN-LOPEZ, A., ARCURI, A., SEGURA, S., AND RUIZ-CORTÉS, A. Black-box and white-box test case generation for restful apis: Enemies or allies? In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)* (2021), IEEE, pp. 231–241.

[44] MATEUS-COELHO, N., CRUZ-CUNHA, M., AND FERREIRA, L. G. Security in microservices architectures. *Procedia Computer Science 181* (2021), 1225–1236.

[45] MITRE. CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection'), 2006. CWE.

[46] MITRE. CWE-94: Improper Control of Generation of Code ('Code Injection'), 2006. CWE.

[47] MYERS, G. J., SANDLER, C., AND BADGETT, T. *The art of software testing*. John Wiley & Sons, 2011.

[48] NEUMANN, A., LARANJEIRO, N., AND BERNARDINO, J. An analysis of public rest web service apis. *IEEE Transactions on Services Computing* (2018).

[49] NEWMAN, S. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[50] NVD. CVE-2024-35451, 2024.

[51] OHM, M., PLATE, H., SYKOSCH, A., AND MEIER, M. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17* (2020), Springer, pp. 23–43.

[52] OLSTHOORN, M., VAN DEURSEN, A., AND PANICHELLA, A. Generating highly-structured input data by combining search-based testing and grammar-based

fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), pp. 1224–1228.

[53] OWASP. Owasp top 10 - 2021.

[54] RODRÍGUEZ, C., BAEZ, M., DANIEL, F., CASATI, F., TRABUCCO, J. C., CANALI, L., AND PERCANNELLA, G. Rest apis: a large-scale analysis of compliance with principles and best practices. In *International Conference on Web Engineering* (2016), Springer, pp. 21–39.

[55] SPADINI, D., ANICHE, M., BRUNTINK, M., AND BACCHELLI, A. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (2017), IEEE, pp. 402–412.

[56] THOMÉ, J., GORLA, A., AND ZELLER, A. Search-based security testing of web applications. In *Proceedings of the 7th International workshop on search-based software testing* (2014), pp. 5–14.

[57] TSAI, C.-H., TSAI, S.-C., AND HUANG, S.-K. Rest api fuzzing by coverage level guided blackbox testing. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)* (2021), IEEE, pp. 291–300.

[58] VELDKAMP, L., OLSTHOORN, M., AND PANICHELLA, A. Grammar-based evolutionary fuzzing for json-rpc apis. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)* (2023), IEEE, pp. 33–36.

[59] WANG, E., CHEN, J., XIE, W., WANG, C., GAO, Y., WANG, Z., DUAN, H., LIU, Y., AND WANG, B. Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications. In *2024 IEEE Symposium on Security and Privacy (SP)* (2024), IEEE, pp. 239–257.

[60] WANG, Y., AND XU, Y. Beyond rest: Introducing apif for comprehensive api vulnerability fuzzing. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses* (2024), pp. 435–449.

[61] ZHANG, M., AND ARCURI, A. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021).

[62] ZHANG, M., AND ARCURI, A. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (may 2023).

[63] ZHANG, M., ARCURI, A., LI, Y., XUE, K., WANG, Z., HUO, J., AND HUANG, W. Fuzzing microservices in industry: Experience of applying evomaster at meituan, 2022.

[64] ZHU, X., WEN, S., CAMTEPE, S., AND XIANG, Y. Fuzzing: A survey for roadmap. *ACM Computing Surveys 54*, 11s (sep 2022).

Part II

Articles

Chapter 1

Search-Based Mock Generation of External Web Service Interactions

Seran, Susruthan, Man Zhang, and Andrea Arcuri

In *International Symposium on Search Based Software Engineering* (pp. 52-66). Cham: Springer Nature Switzerland.

https://doi.org/10.1007/978-3-031-48796-5_4

1.1 Abstract

Testing large and complex enterprise software systems can be a challenging task. This is especially the case when the functionality of the system depends on interactions with other external services over a network (e.g., external REST APIs). Although several techniques in the research literature have been shown to be effective at generating test cases in many different software testing contexts, dealing with external services is still a major research challenge. In industry, a common approach is to *mock* external web services for testing purposes. However, generating and configuring *mock* web services can be a very time-consuming task. Furthermore, external services may not be under the control of the same developers of the tested application.

In this paper, we present a novel search-based approach aimed at *fully automated* mocking external web services as part of white-box, search-based fuzzing. We rely on code instrumentation to detect all interactions with external services, and how their response data is parsed. We then use such information to enhance a search-based approach for fuzzing. The tested application is automatically modified (by manipulating DNS lookups) to rather interact with instances of mock web servers. The search process not only generates inputs to the tested applications, but also it automatically setups responses in those mock web server instances, aiming at maximizing code coverage and fault-finding. An empirical study on 3 open-source REST APIs from EMB, and one 1 industrial API from an industry partner, shows the effectiveness of our novel techniques, i.e., significantly improves code coverage and fault detection.

1.2 Introduction

Enterprise software backends are typically large and complex. Microservice-based software architecture design helps to tackle the challenges in this domain [20]. In a microservice environment, distributed services are interconnected using communication protocols like HTTP(S), Remote Procedure Call (RPC), and Advanced Message Queuing Protocol (AMQP) in a synchronous or asynchronous manner, to execute a common

goal [20]. Likewise, modern web and mobile applications also connect to each other for various reasons, for instance, using services for authentication purposes (e.g., OAuth) or for financial transactions (e.g., Stripe[1] and PayPal[2]). Especially during automated testing, dealing with such external dependencies can be tricky, as what executed in the tested application depends on what returned from those external APIs. Testing for specific error scenarios or specific input data might not be possible in a systematic way if the tester has no control on those external services.

A popular technique in industry to address this limitation is *mocking*. However, for *system-level testing* of web/enterprise applications, instead of using *mock objects*, use of mock external services offers several advantages. For example, the external web service does not even need to be implemented yet, or up and running to execute a test. Furthermore, behavior of the external web service can be extended to test various scenarios (e.g., different returned HTTP status codes). Mocking an external service using a mock HTTP server is a more advanced and possibly less known case. A popular library for the JVM-based applications that can do this is WireMock[3].

In this paper, we provide novel search-based techniques to generate mock web services in a fully automated manner for white-box fuzzing, for the JVM. Using bytecode instrumentation, we can analyze all interactions with external web services during fuzzing, and automatically create mock servers where their responses will be part of the search process. We focus on JavaScript Object Notation (JSON) payloads, where the structure of the responses can be automatically inferred via taint analysis and on how parsing libraries such as Gson and Jackson are used by those tainted values.

Our novel techniques could be adapted and applied to any white-box fuzzer that works on backend web/enterprise applications (or any application that requires communications with external services over a network), for example, RESTful APIs. For the experiments in this paper, we used our open-source fuzzer EvoMaster [10], as not only it gave the best results among fuzzer comparisons [16, 23], but also currently it seems the only fuzzer that supports white-box analyses for this kind of web/enter-

[1] https://stripe.com/
[2] https://paypal.com/
[3] https://wiremock.org/

prise systems [13]. Our extension enables EvoMaster to generate mock external web service with no need of any manual modification of the System Under Test (SUT). Furthermore, we were able to generate self-contained test suites with integrated mocking capabilities to simulate the same production behavior without relying on any external services up and running.

Our novel developed techniques are evaluated on four RESTful APIs (3 open-source from the EMB corpus [11], and 1 industrial), showing statistically significant improvement in code coverage and fault detection. To enable the replicability of our experiments, our extension to EvoMaster is open-source.

The novel contributions of this paper are:

- A novel search-based approach in which HTTP mock servers are automatically instantiated when fuzzing web/enterprise applications.
- A novel use of taint-analysis to automatically infer the syntactic structure of the JSON payloads expected to be returned from these external services.
- A novel integration of search-based test generation, in which both inputs to the tested API and outputs from the mocked services are part of a search-based optimization.
- An open-source extension of the existing search-based, white-box fuzzer EvoMaster.
- An empirical study on both open-source and industrial APIs, providing empirical evidence that our proposed techniques increase line coverage (e.g., up to 6.9% on average on one of the tested APIs) and find 12 more faults.

1.3 Related Work

In industry, it is a common practice to use mocking techniques during unit testing to deal with dependencies [21]. The term "mocking" is typically associated in the literature with *unit testing*. For example, instead of passing as input an instance of a class that

can make a call to an external service or database, a mock object can be used in which each returned value of its method calls can be configured programmatically directly in the tests (without the need to execute the original code). There are different libraries to help to instantiate and configure *mock objects*. For the JVM, popular libraries are for example Mockito,[4] EasyMock,[5] and JMock,[6] which significantly simplify the writing of mock objects.

Although mocking has several practical benefits [21], it can still require a significant time and effort investment when mocks are created manually, as the returned values of each method call on such mocks needs to be specified.

Besides the case of writing unit tests manually, automated unit test generators can be extended to create mock objects as well for the inputs of the classes under test (CUT). An example is the popular EvoSuite [12], which can generate mock object inputs using Mockito [6]. Similarly, Pex can setup mock objects related to the file system APIs [18].

Interactions with the environment can be mocked away if they are executed on method invocations of input objects (as such input objects can be replaced with configurable mocks in the test cases). However, it cannot be mocked away directly if it is the CUT itself that is doing such interactions with the environment. This is a major issue, especially if the CUT is doing operations on the file system (as random files could be created and deleted during the test generation, which could have disastrous consequences). To overcome such a major issue, tools such as EvoSuite can do bytecode instrumentation on the CUT to replace different types of environment operations with calls on a virtual file system [4] and a virtual network [5, 15]. Still, there are major issues in creating the right data with the right structure (e.g., files and HTTP responses) returned from the calls to the virtual environment [15].

For system testing, in which mock objects cannot be typically used as the SUT is executed through the user interfaces (e.g., a GUI or network calls in the case of web services), there is still the issue of how to deal with network calls to external services during testing. Instead of communicating with the actual external services, the SUT

[4] `https://site.mockito.org/`
[5] `https://easymock.org/`
[6] `https://www.jmock.org/`

can be modified (e.g., via parameters to change the hostname/IP address of these services) to speak with a different server, on which the tester has full control on how responses are returned. Simulating/mocking entire external web services is different from mocking the network and dependencies through mock objects, although they share the same goal. From the perspective of the SUT, it is not aware of whether it is communicating with a real service or a mocked one, as it will still use the same code to make network calls (e.g., HTTP over TCP) and the same code to read and parse the responses. There are several tools and libraries available in industry which help to setup and run a mocked external web service with less effort, such as Mockoon,[7] Postman,[8] and WireMock.[9] All of them share common functionalities to create configurable mock HTTP(S) web services for testing.

Creating handwritten mock external web services can be tedious and time-consuming. Furthermore, if the third-party interactions are executed using a provided software development kit (SDK), unless the knowledge about the interactions is available or the SDK is reverse-engineered, the process becomes convoluted. A complementary approach is to use *record&replay* where each request and response is captured using a pass-through proxy by the mock server rather than configuring them manually. Later on, the captured information will be used to initiate the mock server. However, the external web service should be owned by the same developers of the SUT to be able to emulate more use cases (especially error scenarios), apart from the common "happy path" scenarios. When the system scales up in size, handwritten mock web services is not a scalable approach. However, record and replay would still be feasible.

The work done in [5, 15] is perhaps the closest in the literature to what we present in this paper. However, there are some significant major differences. First, we do not use a limited, artificial virtual network, but rather make actual HTTP calls using external mocked web services (using WireMock). To be able to achieve this automatically, we have to overcome several challenges, especially when dealing with SSL encryption, and when multiple external services use the same TCP port. Furthermore, to enable ef-

[7] `https://mockoon.com/`
[8] `https://www.getpostman.com/`
[9] `https://wiremock.org/`

fective system testing, we need to automatically generate mocked responses with valid syntax (e.g., in JSON representing valid Data Transfer Objects (DTO) for the mocked service), regardless of whether any formal schema is present or not (e.g., the work in [15] requires specifying XSD files manually with the schema of the response messages). Furthermore, while the work in [15] was evaluated only on an artificial class, we show that our technique can scale on real systems (including an industrial API).

To the best of our knowledge, no technique exists in the scientific literature that can fully automate the process of mock generation for handing external web service dependencies. Moreover, fully automated mock generation of external web services for system level white-box testing is not a topic that has received much attention in the research literature, despite its importance in industry (e.g., considering all the existing popular libraries/tools such as WireMock and Postman).

EvoMaster is an open-source tool used for fuzzing web services using search-based techniques with white-box and black-box fuzzing [10, 2]. EvoMaster provides client-side drivers to enable white-box fuzzing and requires no code modification on the SUT, as instrumentation is done automatically. However, the driver requires writing the necessary steps to cover the three main stages of the API's lifecycle, such as start, reset, and stop. Furthermore, EvoMaster features different search algorithms (e.g., MIO [1] extended with adaptive-hypermutation [22]) and fitness functions (e.g., using advanced testability transformations [8] and heuristics based on the SQL commands executed by the SUT [7]) to generate system level test suites.

1.4 HTTP Mocking

The objective of our work is to enable automated mocking and configuration of external web services during search-based test generation. We focus on HTTP services that use JSON for the response payloads, as those are the most common types of web services in industry [19]. To achieve this goal, a good deal of research, and technical challenges, need to be addressed, as discussed next in more details.

1.4.1 Instrumentation

During the search, we need to automatically detect the hostnames/IP addresses and ports of all the external services the SUT communicates with. To achieve this, we applied a form of *testability transformation* [14], relying on the infrastructure of EvoMaster to apply *method replacements* for common library methods [8] (e.g., the APIs of JDK itself). We extended the white-box instrumentation of EvoMaster with new replacement methods for APIs related to networking. When classes are loaded into the JVM, method calls towards those APIs are replaced with our methods. Those can track what inputs were used, and then call the original API without altering the semantic of the SUT.

We created method replacements for Java network classes, such as `java.net.InetAddress`, `java.net.Socket`, and `java.net.URL`. This enables us to collect and manipulate various information related to the connection such as hostname, protocol, and port from the different layers of the Open Systems Interconnection (OSI) model.

Using the first call made to the external web service, we can capture (and alter) the DNS information about the external web service through the instrumented `java.net.InetAddress`. After the IP address is resolved for the given hostname, when the SUT tries to initiate the `Socket` connection we can capture the port information as well. If the connection is rather created through a `URL` object, such information can be derived directly there.

To make this discussion more clear, let us consider the example in Figure 1.1. This is a small artificial example of a REST endpoint written in SpringBoot making an HTTP call towards an external service using a `URL` object. If the request is successful with the response "`HELLO THERE!!!`", the application will respond with an HTTP 200 status code with the string body "`YES`", otherwise with a "`NOPE`".

The `openConnection()` call executed at Line 5 will establish the connection with the remote destination (at the fictional `hello.there:8123`), and it will return a `java.net.URLConnection` to enable to send and receive data to such external end-

```
1 @GetMapping(path = ["/string"])
2 fun getString() : ResponseEntity<String> {
3
4    val url = URL("http://hello.there:8123/api/string")
5    val connection = url.openConnection()
6    connection.setRequestProperty("accept", "application/json")
7    val data = connection.getInputStream()
8                         .bufferedReader()
9                         .use(BufferedReader::readText)
10
11   return if (data == "\"HELLO THERE!!!\""){
12        ResponseEntity.ok("YES")
13   } else{
14        ResponseEntity.ok("NOPE")
15   }
16 }
```

Figure 1.1: Small example of a REST endpoint written in SpringBoot with Kotlin, making an HTTP call towards an external service using a `URL` object.

point. Our instrumentation will replace the call `url.openConnection()` with our custom `URLClassReplacement.openConnection(url)`. Inside this function, we can get all information about the HTTP connection, and create a new connection where instead of connecting to `hello.there:8123` we connect to an instance of WireMock we control.

Note that this kind of instrumentation is applied to all classes loaded in the JVM, and not just in the business logic of the SUT. For example, if the SUT is using a client library to connect to the external services, this approach will still work. An example in our case study (Section 1.5) is *catwatch*, where it uses the Java library `org.kohsuke:github-api`, which internally connects to `https://api.github.com`.

However, during testing, we want to avoid messing up with the connections with controlled services, such as databases. For this reason, we do not apply any modification when the SUT connects to services running on the hostname `localhost`. This is not a problem when dealing with external services running on internet (e.g., `https://api.github.com`). However, in microservice architectures, the external services are often running on the same host. This is, for example, an issue for the API *cwa-verification* used in our case study (Section 1.5). In this case, the SUT needs to be started with `localhost` replaced with any other hostname, possibly via configuration

options. In the case of *cwa-verification*, this was easily achieved by manually overriding the option `-cwa-testresult-server.url`.

1.4.2 The Mock Server

Once we collect the information on hostnames, protocols, and ports of the external services, we have enough information to initiate mock servers (one per external service).

Writing a mock server that can handle HTTP requests is a major engineering effort. Considering the large amount of work that would be needed to write our own mock server, we rather decided to use an existing mock server library, with a wide use in industry. As the implementation of our techniques is targeting the JVM (e.g., for programs written for example in Java and Kotlin), we needed a mock server capable of running inside the JVM and be programmatically configurable. We chose to use WireMock[10] for this purpose, since it satisfies all our requirements.

WireMock can be configured manually (e.g., by writing all the reposes before staring the server) or programmatically (e.g., during runtime). Apart from that, WireMock supports record and replay as well to configure the mock server. For our purpose, we have used WireMock programmatically. The instrumentation allowed us to reroute the external web service subsequent requests to the respective WireMock server.

Each of the responses can be defined using stubs inside WireMock. A stub can be configured using a static URL path (e.g., `/api/v1/`) or a regular expression-based path (e.g., `/api/v1/.*`). Furthermore, a stub can be extended using various parameters to match a request. The default behavior of WireMock is to throw an exception if a request pattern is not configured. This caused problems during the search. To overcome the difficulties, we set WireMock to return a response with an HTTP 404 for all requests if a stub is not present.

Once initiated, all the requests will be redirected to the respective WireMock server using instrumentation. After a test case is executed, we can query WireMock to retrieve the captured request patterns.

[10] `https://wiremock.org/`

When initiating WireMock servers, we faced difficulties regarding TCP ports. SUT may connect to various external web services using the same TCP port (e.g., HTTP 80 or HTTPS 443). During the search, we have control over the SUT through instrumentation. However, it was difficult to maintain the same behavior in the generated test cases (e.g., JUnit files) where no bytecode instrumentation is applied. This means generated test cases should behave the same way without any modification required, like during the search.

Different operating systems (OS) handle TCP ports in different ways, especially regarding available ephemeral TCP ports and the *TCP Time Wait* delays. *TCP Time Wait* defines the amount of time it will take to release a TCP port once it is freed from the previous process. Each of the operating systems has a different default value for *TCP Time Wait*. This is a major issue for empirical research, when running several experiments on the same machine in parallel.

Using different addresses from the loopback subnet range (e.g.,*127.0.0.0/8*) is an easy and elegant solution to overcome the challenges we faced with the limitation of available ephemeral TCP ports to initiate WireMock servers. Moreover, it eradicated the dependency on *TCP Time Wait* delays. Furthermore, this allowed us to create and manage mock servers with no code modification required from the SUT. However, some operating systems might handle loopback addresses differently. For example, *macOS* does not allow to access addresses from this range besides *127.0.0.1* by default. So, other addresses from the range should be configured as an alias on the respective network interface (in most cases, it is *lo0*). However, we have not seen this kind of problem when running experiments on *Linux* and *Windows*.

The decision to pick the loopback IP address to host the mock server happens automatically. However, in our tool extension it can be configured using various options. Three different initial local IP address allocation strategies have been developed. This helped to avoid conflicts during running experiments in parallel and to have more control over address allocation (e.g., in the generated test cases). Because of the space limitation, we cannot go into details about explaining each strategy.

Furthermore, in each setting, some of the loopback addresses will be skipped. This

includes network addresses *127.0.0.0*, *127.0.0.1* and the broadcast address of the range *127.255.255.255*, to avoid unanticipated side effects.

As mentioned earlier, to ensure the same behavior in the generated test cases as the search, we relied on *Java Reflection*, especially needed when hostnames are hard-coded in third-party libraries and cannot be easily modified by the user (e.g., like the case of `https://api.github.com` in *catwatch*). We managed to modify the JVM Domain Name System (DNS) cache through reflection in the generated test cases by using the `dns-cache-manipulator` library from Alibaba. This way, in the generated JUnit tests, we can still remap a hostname such as `api.github.com` to a loopback address where a WireMock instance is running. However, one (arguably minor) limitation here is that we cannot handle in this way the cases in which IP addresses are hard-coded without using any hostname (e.g., using *140.82.121.6* instead of `api.github.com`). But the use of hard-coded IP addresses does not seem a common practice.

1.4.3 Requests and Responses

As previously mentioned, the instantiated WireMocks will run with a default response HTTP 404 for all the request patterns. However, mocking the external web services with different correct responses is necessary to achieve better code coverage. Recall the example in Figure 1.1, where the result of an `if` statement depends on whether the external service returns the string "`HELLO THERE!!!`". To reach this point of execution, the mock server should give the response as expected. It would be extremely unlikely to get the right needed data at random. Performance improvement of the search requires better heuristics in a case like this.

Besides body payloads, there could be a possibility for SUT to check for other HTTP response parameters as well (e.g., HTTP status code and headers). To maximize code coverage in the SUT (and so indirectly increase the chances of detecting faults), there is the need to have various HTTP responses in each stage of the search to cover all these possible cases.

To address these issues, we have extended the fuzzer engine of EvoMaster to

create mock responses. Besides evolving the parameters and payloads of the REST API calls towards the SUT, now our EvoMaster extension can also evolve the data in the mock responses.

Internally, each REST API call "action" in a test will have associated actions related to setup these mock. Once a test is executed, and its fitness evaluation is computed, through the instrumentation we gather information about the interactions the SUT had with the external services from the respective WireMock servers. In the subsequent steps of the search, when the test is selected again for mutation, these newly created actions will be mutated, and the respective WireMock server will have a new response for the request pattern besides the default `HTTP 404`. Initially, the genotype of these WireMock setup actions will contain mutable genes to handle the returned HTTP status code and an optional body payload (for example, treated as random strings). Throughout the search, actions will enhance the respective WireMock responses by mutating those genes, and then collecting their impact on the fitness function.

It is possible to easily randomize some parameters in a typical HTTP response while searching, such as HTTP status code (e.g., typically range is from 100 to 599). But, creating correct values in parameters like headers and response bodies is not a straightforward task. With no information available at the beginning of the search, it is necessary to find a way to gather this information to create a response schema. For example, when the SUT expects a specific JSON schema as a response, sending a random string as body payload will lead to throwing an exception in the data parsing library of the SUT.

In case the schema is available in a commonly used format such as OpenAPI/Swagger, it is possible to infer it from that. However, it is not always possible. An alternative, more general solution is to use instrumentation to analyze how such body payloads are parsed by the SUT.

The `JSON` is a common choice as data transfer format for the interactions between web services [19]. Use of Data Transfer Objects (DTO) is a known practice in most of the statically typed programming languages to represent the schema in code. When a `JSON` payload is received from an HTTP request, a library can be used to parse such

text data into a DTO object. By instrumenting the relevant libraries, at runtime during the test evaluation we can analyze these DTOs to infer the structure of the schema of these JSON messages.

On the JVM, the most popular libraries for JSON parsing are *Gson* and *Jackson* [17]. For these two libraries, we provide method replacements for their main entry points related to parsing DTOs, for example `<T> T fromJson(String json, Class<T> classOfT)`. In these method replacements, we can see how any `JSON` string data is parsed to DTOs (analyzing as well all the different Java annotations in these libraries used to customize the parsing, for example `@Ignored` and `@JsonProperty`). This information can then be fed back to the search engine: instead of evolving random strings, EvoMaster can then evolve JSON objects matching those DTO structures.

There is one further challenge that needs to be addressed here: how to trace a specific `JSON` text input to the source it comes from. It can come from an external web service we are mocking, but that could use different DTOs for each of its different endpoints. Or, it could come from a database, or as input to the SUT, and have nothing to do with any of our WireMock instances. The solution here is to use *taint analysis*. In particular, we extend the current input tracking system in EvoMaster [9]. Each time a string input is used as body payload in the mocked responses, it will not be a random string, but rather a specific tainted value matching the regex `_EM_\d+_XYZ_`, e.g., `_EM_0_XYZ_`. In each of our method replacements (e.g., for `fromJson`) we check if the input string does match that regular expression. If so, we can check the genotype of the executed test case to see where that value is coming from. The gene representing that value is then modified in the next mutation operation to rather represent a JSON object valid for that DTO.

During the search, modifications/mutations to an evolving test case might lead the SUT to connect to new web services, or not connect anymore to any (e.g., if a mutation leads the test case to fail input validation, and then the SUT directly returns a 400 status code without executing any business logic). If no matching request exists to the existing actions after a fitness evaluation, it will be disabled. The rest will be marked as active for further use. If an action representing a call to the SUT is deleted, then all

Table 1.1: Descriptive statistics of case studies. Note that, for each case study, #Endpoints represents several endpoints, #Classess represents numerous classes and #File LOCs represents numerous lines of code (LOC).

SUT	#Endpoints	#Classes	#File LOCs
catwatch	23	106	9636
cwa-verification	5	47	3955
genome-nexus	23	832	64339
ind1	54	115	7112
Total	105	1100	85042

the WireMock stubs associated with it are deleted as well.

1.5 Empirical Study

To evaluate the effectiveness of our novel techniques presented in this paper, we carried out an empirical study to answer the following research question:

What is the impact on code coverage and fault-finding of our novel search-based approach for external service mocking?

1.5.1 Experiment Setup

To evaluate our approach, we conducted our empirical study with three case studies from EMB [11] and one industrial case study (i.e., *ind1*). EMB is an open-source corpus composed of open-source web/enterprise APIs for scientific research [11].

To assess the effectiveness of our mock generation, we selected from EMB all the REST APIs which require to communicate with external web services for their business logic (i.e., *catwatch*, *cwa-verification*, and *genome-nexus*). Descriptive statistics of the selected case studies are reported in Table 1.1. In total, 105 endpoints and 85*k* lines of codes were employed in these experiments. Note that these statistics only concern code for implementing the business logic in these APIs. The code related to third-part libraries (which can be millions of lines [11]) is not counted here.

catwatch is a web application which allows fetching GitHub statistics for GitHub accounts, processes, and saves the data in a database and provides them via a RESTful API. The application heavily relies on GitHub APIs to perform its core functions.

cwa-verification is a component of the official Corona-Warn-App for Germany. This case study is a part of a microservice-based architecture which contains various other services, including mobile and web apps. The tested backend server relies on other services in the microservice architecture to complete its tasks.

genome-nexus is an interpretation tool which aggregates information from various other online services about genetic variants in cancer.

ind1 is a component in an e-commerce system. It deals with authentication using an external Auth0 server, and it processes monetary transactions using Stripe[11].

To assess our novel approach, we integrated it into our open-source, white-box fuzzer EvoMaster. Then we conducted experiments to compare the baseline and our novel proposed approach, i.e., *Base* refers to EvoMaster with its default configuration, and *WM* refers to our approach which enables our handling of mock generation with EvoMaster. Considering the randomness nature of search algorithms, we repeated each setting 30 times using the same termination criterion (i.e., 1 hour), by following common guidelines for assessing randomized techniques in software engineering research [3].

Interactions with real external services are often non-deterministic, as the external services might not be accessible and change their behavior at any time. For instance, with a preliminary study, we found that *catwatch* communicates with Github API, but there exists a rate limiter (a typical method to prevent DoS attacks) to control the access to this API[12] based on IP address, e.g., up to 60 requests per hour for unauthenticated requests. Considering the network setup of our university, we may share the same IP address with all our colleagues and students (e.g., when behind a NAT router). Then, such rate limit configuration will strongly affect results on this study, e.g., depending on the time of the day in which the experiments are running, in some experiments

[11] `https://stripe.com/`

[12] `https://docs.github.com/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28`

we might be able to fetch data from GitHub, but not in others. As this can lead to completely unreliable results when comparing techniques, we decided to run all the experiments with internet disabled. This also provides a way to evaluate our techniques in the cases in which the external services are not implemented yet (e.g., at the beginning of a new project) or are temporarily down.

With two settings (i.e., Base and WM) on four case studies using 1 hour as search budget, 30 repetitions of the experiments took 240 hours (10 days), i.e., $2 \times 4 \times 30 \times 1$. All experiments were run on the same machine, i.e., HP Z6 G4 Workstation with Intel(R) Xeon(R) Gold 6240R CPU @2.40GHz 2.39GHz, 192 GB RAM, and 64-bit Windows 10 OS.

1.5.2 Experiment Results

To answer our research question, we report line coverage and number of detected faults on average (i.e., `mean`) with 30 repetitions achieved by Base and WM, as shown in Table 1.2. We employed a Mann–Whitney U test (i.e., p-value) and Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) to perform comparison analyses between Base and WM in terms of line coverage and fault detection. With Mann–Whitney U test, if p-value is less than the standard significance level (i.e., 0.05), it indicates that the two compared groups (i.e., Base and WM) have statistically significant differences. Otherwise, there is no enough evidence to support the claim the difference is significant. Comparing WM with Base, the Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) measures how likely WM can perform better than Base. $\hat{A}_{12} = 0.5$ represents no effect. If $\hat{A}_{12}$ surpasses 0.5, it indicates that WM has more chances to achieve better results than Base.

Based on the analysis results reported in Table 1.2, compared to Base, we found that WM achieved consistent improvements on both metrics, i.e., line coverage and detected faults. The results show that, for all the four selected APIs, our approach (i.e., WM) significantly outperformed Base with low p-values and high $\hat{A}_{12}$. Such results demonstrate effectiveness of our approach with white-box heuristic and taint analysis to guide mock object generation.

Table 1.2: Results of line coverage and detected faults achieved by Base and WM. We also report pair comparison results between Base and WM using Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) and Mann–Whitney U test (i.e., p-value at significant level 0.05).

	Line Coverage %				Detected Faults %			
SUT	Base	WM	$\hat{A}_{12}$	p-value	Base	WM	$\hat{A}_{12}$	p-value
catwatch	46.4	53.3	**0.99**	< 0.001	41.5	45.4	**0.82**	< 0.001
cwa-verification	57.4	59.9	**0.95**	< 0.001	12.3	13.8	**0.87**	< 0.001
genome-nexus	27.9	30.3	**1.00**	< 0.001	21.2	22.0	**0.71**	< 0.001
ind1	14.2	15.1	**0.70**	0.002	59.7	65.1	**0.97**	< 0.001
Average	36.5	39.7	0.91		33.7	36.6	0.84	

For *genome-nexus*, we found that most of the interactions from the SUT with external services is to fetch data, then save the data into a database (i.e., MongoDB). In this case, it is not necessary to extract the data when fetching them, and such data extraction can be performed when saving data into the database. Code snippet of an interaction and data extraction in *genome-nexus* is shown as below (complete code can be found in `BaseCachedExternalResourceFetcher.java`[13]).

```
1 rawValue = this.fetcher.fetchRawValue(this.buildRequestBody(subSet));
2 ...
3 rawValue = this.normalizeResponse(rawValue);
4 ...
5 List<T> fetched = this.transformer.transform(rawValue, this.type);
6
```

Based on the code, line 1 does fetch to get raw data. By checking the implementation of `fetchRawValue` (e.g., `BaseExternalResourceFetcher.java`[13]), a general type (such as `BasicDBList`[14]) is provided. With such type info, we can only know it is a list. The actual data extraction based on the raw value is performed later in Line 5 before saving it into database. Our current handling does not support such response schema extraction yet. It can be considered as a future work.

Unfortunately, for reasons of space, we cannot discuss in more details the results

[13] `https://github.com/EMResearch/EMB`

[14] `https://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBList.html`

obtained on the other APIs.

***RQ**: Our approach with white-box heuristics and taint analysis demonstrates its effectiveness to guide mock object generation, increasing code coverage and fault detection. Such improvements for a search-based fuzzer (i.e., EvoMaster) are statistically significant on all the four selected APIs.*

1.6 Conclusion

In this paper, we have provided a novel search-based approach to enhance white-box fuzzing with automated mocking of external web services. Our techniques have been implemented as an extension of the fuzzer EvoMaster [10], using WireMock for the mocked external services. Experiments on 3 open-source and 1 industrial APIs show the effectiveness of our novel techniques, both in terms of code coverage and fault detection.

To the best of our knowledge, this is the first work in the literature that provides a working solution for this important problem. Therefore, there are several avenues for further enhancements of our techniques in future work. An example is how to effectively deal with APIs that have a 2-phase parsing of JSON payloads (like in the case of *genome-nexus* in our case study).

Our tool extension of EvoMaster is released as open-source, with a replication package for this study currently available on GitHub[15].

Acknowledgment

This work is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

[15] `https://github.com/researcher-for/es-mocking`

Bibliography

[1] ARCURI, A. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology 104* (2018), 195–206.

[2] ARCURI, A. Automated black-and white-box testing of restful apis with evomaster. *IEEE Software 38*, 3 (2020), 72–78.

[3] ARCURI, A., AND BRIAND, L. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR) 24*, 3 (2014), 219–250.

[4] ARCURI, A., FRASER, G., AND GALEOTTI, J. P. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (2014), pp. 79–90.

[5] ARCURI, A., FRASER, G., AND GALEOTTI, J. P. Generating tcp/udp network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 155–165.

[6] ARCURI, A., FRASER, G., AND JUST, R. Private api access and functional mocking in automated unit test generation. In *2017 IEEE international conference on software testing, verification and validation (ICST)* (2017), IEEE, pp. 126–137.

[7] ARCURI, A., AND GALEOTTI, J. P. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM) 29*, 4 (2020), 1–31.

[8] ARCURI, A., AND GALEOTTI, J. P. Enhancing Search-based Testing with Testability Transformations for Existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021), 1–34.

[9] ARCURI, A., AND GALEOTTI, J. P. Enhancing search-based testing with testability transformations for existing apis. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021), 1–34.

[10] ARCURI, A., GALEOTTI, J. P., MARCULESCU, B., AND ZHANG, M. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software 6*, 57 (2021), 2153.

[11] ARCURI, A., ZHANG, M., GOLMOHAMMADI, A., BELHADI, A., GALEOTTI, J. P., MARCULESCU, B., AND SERAN, S. EMB: A curated corpus of web/enterprise applications and library support for software testing research. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (2023), IEEE, pp. 433–442.

[12] FRASER, G., AND ARCURI, A. EvoSuite: automatic generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)* (2011), pp. 416–419.

[13] GOLMOHAMMADI, A., ZHANG, M., AND ARCURI, A. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* (aug 2023).

[14] HARMAN, M., HU, L., HIERONS, R., WEGENER, J., STHAMER, H., BARESEL, A., AND ROPER, M. Testability transformation. *IEEE Transactions on Software Engineering 30*, 1 (2004), 3–16.

[15] HAVRIKOV, N., GAMBI, A., ZELLER, A., ARCURI, A., AND GALEOTTI, J. P. Generating unit tests with structured system interactions. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)* (2017), IEEE, pp. 30–33.

[16] KIM, M., XIN, Q., SINHA, S., AND ORSO, A. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2022), ISSTA 2022, Association for Computing Machinery, p. 289–301.

[17] MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)* (2012), IEEE, pp. 177–182.

[18] MARRI, M. R., XIE, T., TILLMANN, N., DE HALLEUX, J., AND SCHULTE, W. An empirical study of testing file-system-dependent software with mock objects. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on* (2009), pp. 149–153.

[19] NEUMANN, A., LARANJEIRO, N., AND BERNARDINO, J. An analysis of public rest web service apis. *IEEE Transactions on Services Computing* (2018).

[20] NEWMAN, S. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[21] SPADINI, D., ANICHE, M., BRUNTINK, M., AND BACCHELLI, A. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (2017), IEEE, pp. 402–412.

[22] ZHANG, M., AND ARCURI, A. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021).

[23] ZHANG, M., AND ARCURI, A. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (may 2023).

Chapter 2

Handling Web Service Interactions in Fuzzing with Search-Based Mock-Generation

Seran, Susruthan, Man Zhang, Onur Duman, and Andrea Arcuri

ACM Transactions on Software Engineering and Methodology (2025)

https://doi.org/10.1145/3731558

2.1 Abstract

Testing large and complex enterprise software systems can be a challenging task. This is especially the case when the functionality of the system depends on interactions with other external services over a network (e.g., external web services accessed through REST API calls). Although several techniques in the research literature have been shown to be effective at generating test cases in a good number of different software testing contexts, dealing with external services is still a major research challenge. In industry, a common approach is to *mock* external web services for testing purposes. However, generating and configuring *mock* web services can be a very time-consuming task, e.g., external services may not be under the control of the same developers of the tested application, making it challenging to identify the external services and simulate various possible responses.

In this paper, we present a novel search-based approach aimed at *fully automated* mocking of external web services as part of white-box, search-based fuzzing. We rely on code instrumentation to detect all interactions with external services, and how their response data is parsed. We then use such information to enhance a search-based approach for fuzzing. The tested application is automatically modified (by manipulating DNS lookups) to rather interact with instances of mock web servers. The search process not only generates inputs to the tested applications but also automatically configures responses in those mock web server instances, aiming at maximizing code coverage and fault-finding. An empirical study on four open-source REST APIs from EMB, and one industrial API from an industry partner, shows the effectiveness of our novel techniques (i.e., in terms of line coverage and fault detection).

2.2 Introduction

Microservice-based software architecture design helps to tackle the challenges in developing large and complex enterprise software programs. In a microservice environment, relatively small services are interconnected using communication protocols

like Hypertext Transfer Protocol (HTTP), Remote Procedure Call (RPC), and Advanced Message Queuing Protocol (AMQP) in a synchronous or asynchronous manner, to execute a common goal. Such independent microservices help to construct a complex system. For example, one microservice might represent payment processing, another authentication, and another content delivery, but altogether they might constitute an entire video streaming service. Even though the microservice architecture disentangles the development process, it adds more complexity to system-level software testing [42]. Likewise, modern web and mobile backends also connect to each other for various reasons, for instance, using services for authentication purposes (e.g., OAuth[1]) or for financial transactions (e.g., Stripe[2] and PayPal[3]).

Especially during automated testing, dealing with such external dependencies can be tricky since what is executed in the system under test (SUT) purely depends on what is returned from those external web services. Also, testing for specific error scenarios or specific input data might not be possible in a systematic way if the tester has no control on those external services.

In order to deal with such dependencies, *mocking* is a popular technique used in industry [49, 9]. In the existing literature, the term "mocking" is commonly associated with the usage of *mock objects* in *unit testing* [49]. Mockito[4] is one of the popular libraries used to create such *Mock Objects* for the Java Virtual Machine (JVM) based applications. On the other hand, for *system-level testing* of web/enterprise applications, instead of using *mock objects*, the usage of external service mock servers offers several advantages. For example, the external web service does not even need to be implemented yet or up and running to execute a test. Furthermore, the behavior of the external web service can be extended to test various scenarios (e.g., different returned HTTP status codes). One of the most popular libraries for JVM-based applications that can achieve this is WireMock.[5]

In general, testers manually configure mock servers by specifying the payload that

[1] `https://oauth.net`
[2] `https://stripe.com/`
[3] `https://paypal.com/`
[4] `https://site.mockito.org/`
[5] `https://wiremock.org/`

should be returned for every possible request. This can be tedious and time-consuming [2], especially if the SUT interacts with several external services. A complementary approach is to use *record-and-replay*, where each request and response are captured using a pass-through proxy to initiate the mock server rather than configuring it manually [31]. Even though this approach tackles the problem to an extent, it falls short in dealing with the scalability issue as the system scales up in size. Furthermore, this approach is less flexible, as it may not be possible to test edge cases in this way. Another major issue is that often third-party interactions are executed using a provided software development kit (SDK) or a client library. The actual network calls and their input/output objects might be hidden away from the user. Consequently, the developer may not possess any knowledge about these network communications unless the library or SDK is reverse-engineered, and this can be very time-consuming. Consequently, the process of handwritten mocks becomes further complex in these cases. By taking all of those gaps into consideration, we aim to develop techniques to generate mock objects in an automated manner.

In this paper, we present novel search-based, white-box fuzzing techniques to automatically generate mock external web services for the JVM-based applications. White-box testing is dependent on the programming language, as the code must be analyzed. The JVM-based languages, especially Java and Kotlin, are two of the most widely used in industry. Despite the popularity of Java and Kotlin, there are other languages/runtimes that are widely used to develop web APIs, such as JavaScript, Go, and .NET. However, supporting diverse programming languages for white-box testing is a significant undertaking, which is often viewed solely as technical work by researchers. Therefore, for technical and practical reasons our objective is to focus solely on case studies based on Java and Kotlin. Moreover, JVM capabilities such as Java Reflection and bytecode instrumentation enable us to implement our approach. Notably, bytecode instrumentation is a technique employed to modify or analyze the bytecode of a program at runtime (e.g., JVM and .NET) or during the compilation process. Through bytecode instrumentation, we can automatically analyze all interactions with external web services at runtime, and we can collect heuristics (i.e., response schema from the

used Data Transfer Objects (DTO)). Similarly, Java Reflection is a technique used in Java that allows a program to inspect and manipulate the program at runtime. By utilizing the collected information using both techniques, we can create mock web services with proper information automatically. To enable our novel techniques, we analyze all interactions with external services using bytecode instrumentation and automatically create mock web servers where responses from those are utilized as part of search-based testing.

Regarding data transfer formats, we focus on JavaScript Object Notation (JSON) payloads, as JSON-based DTOs are one of the widely used formats in the industry [41, 46]. Furthermore, in JSON-based DTOs, the structure of responses can be inferred automatically by instrumenting parsing libraries (e.g., Gson[6] and Jackson[7]) and by preforming taint analysis. Taint analysis is a software program analysis technique that monitors data flow within a program, facilitating the identification of software faults, particularly software vulnerabilities and data leaks. Nevertheless, this approach has the potential to be extended to other schema formats as well, like XML.

Our novel techniques can be adapted and applied to any white-box fuzzer that works on backend web/enterprise applications (or any application that requires communications with external services over a network), like for example RESTful APIs. For the experiments in this paper, to reduce the workload required to assess our novel techniques, we chose to extend an existing search-based fuzzer instead of developing a new one from scratch. We opted an open-source fuzzer EvoMaster [13], since it performs best compared to other black-box fuzzers [30, 54]. Moreover, to the best of our knowledge EvoMaster is the only open-source fuzzer that supports white-box testing for web applications [25]. Our extension enables EvoMaster to generate mock external web services with zero modification required to the System Under Test (SUT). Furthermore, we generate self-contained test suites with integrated mocking capabilities to simulate the same production behavior without the need for external services to be up and running.

[6] https://github.com/google/gson
[7] https://github.com/FasterXML/jackson

Our techniques are evaluated on five RESTful APIs (four open-source from the EMB[8] corpus [16] and one industrial), showing statistically significant improvement in code coverage and fault detection. To enable the replicability of our experiments, our extension to EvoMaster is open-source,[9] which includes as well the experiment scripts used in the different empirical studies.[10]

The contributions of this paper are:

- A search-based approach in which HTTP mock servers are automatically instantiated when fuzzing web/enterprise applications.
- A use of taint-analysis to automatically infer the syntactic structure of the JSON payloads expected to be returned from these external services.
- An integration of search-based test generation, in which both inputs to the tested API and outputs from the mocked services are part of a search-based optimization.
- An open-source extension of the existing search-based white-box fuzzer EvoMaster.
- An empirical study on both open-source and industrial APIs, providing empirical evidence that our proposed techniques increase line coverage (e.g., up to $+20\%$ on average on one of the tested APIs) and find more faults (e.g., 8.6 more on average).

This article is an extension of a conference article [48]. In this extension, to further improve performance we present new techniques, such as *Harvester* in Section 2.5.6. We also improved the handling of heuristics related to external web service information in Section 2.5.2. Additionally, to better analyze the effectiveness of our novel techniques, we conducted new experiments on artificial APIs (discussed in Section 2.7.2), and experiments with an additional open-source API answering two new research questions, which we discuss in detail under Section 2.7.

[8] https://github.com/WebFuzzing/EMB
[9] https://github.com/WebFuzzing/EvoMaster
[10] https://zenodo.org/records/10932122

This article is structured as follows. Section 2.3 delves into the related work. The motivation and background of the work is discussed in Section 2.4. The details regarding automated mock external web service generation are discussed in Section 2.5. Section 2.6 contains the specifics of how we ensure that the mock external web services maintain the consistent behavior in the self-contained test suites as they do during the search. Section 2.7 includes the details of the empirical study, along with the results and their discussion. Potential threats to the validity are analyzed in Section 2.8. Finally, Section 2.9 concludes the article and discusses future work.

2.3 Related Work

2.3.1 Fuzzing

Fuzzing is a software testing technique that uses random or invalid inputs to be given to the software to analyze its behavior [39]. Fuzzing is one of the most effective methods to detect software faults (in particular security-related faults) and to achieve higher code coverage [22, 34, 59]. There are several success stories about fuzzing techniques, in particular, to test parser libraries written in C/C++ [38], with popular tools such as AFL [1], as well as network devices [58, 45]. Recently, large and complex interconnected enterprise software systems with millions of lines of code have been tested efficiently using fuzzing techniques [55]. Particularly, white-box and black-box fuzzing of REST-based web services has gained a major amount of attention among researchers lately [25, 30, 54]. White-box fuzzers have been proven to be highly effective in numerous instances [23, 37, 5]. In white-box fuzzing, internal details of the SUT, such as its source code, binaries, or bytecode, will be accessed. This information can be used to design heuristics to improve the search and produce better test cases with better line coverage and fault detection capabilities. By leveraging the advantages of search-based white-box fuzzing, in this paper we present techniques to automatically generate mock external web services to further improve fault-finding and code coverage.

2.3.2 Mocking

In industry, it is a common practice to use mocking techniques during unit testing to deal with dependencies. The use of *mock objects* is one of the common practices for handling dependencies in *unit testing* [32, 51, 40, 49]. A mock object can be used to represent the response to a call made to an external service or database. Especially in tests, this can be programmatically configured without executing the original code. Different libraries help instantiate and configure *mock objects*. For JVM-based applications, there exist libraries that significantly simplify the writing of mock objects [40], such as Mockito,[4] EasyMock,[11] and JMock.[12]

Mocking helps to expedite software testing and allows testing of the functionalities individually [49]. Although mocking has practical benefits, it can still require a significant time and effort investment when mocks are created manually, as the returned values of each method call on such mocks need to be specified.

Besides the case of writing unit tests manually, automated unit test generators can be extended to create mock objects as well for the inputs of the classes under test (CUT). An example is the popular EvoSuite [21], which can generate mock object inputs using Mockito [9]. Similarly, Pex can set up mock objects related to the file system APIs [36].

Interactions with the environment can be mocked away if they are executed on method invocations of input objects (as such input objects can be replaced with configurable mocks in the test cases). However, it cannot be mocked away directly if it is the CUT itself that is doing such interactions with the environment. This is a significant concern, especially if the CUT is performing operations on the file system (as random files could be created and deleted during the test generation, which could have severe consequences). To overcome such a major issue, tools such as EvoSuite can do bytecode instrumentation on the CUT to replace different types of environment operations with calls on a virtual file system [7] and a virtual network [8, 28]. Still, there are major issues in creating the right data with the right structure (e.g., files and HTTP responses) returned from the calls to the virtual environment [28].

[11] `https://easymock.org/`
[12] `https://www.jmock.org/`

In system-level testing, mock objects cannot be directly used since the SUT is executed through the user interfaces (e.g., a GUI or network calls in the case of web services). Consequently, there is still the issue of how to deal with network calls to external services during testing. During testing, instead of communicating with the actual external services, the SUT should be modified to communicate with a mock server, on which the tester has full control over how responses are returned. This may potentially help with the testing of additional scenarios.

Mocking entire external web services is different from mocking the network and dependencies through mock objects, although they share the same goal. From the perspective of the SUT, it is not aware of whether it is communicating with a real or mocked service, as it will still use the same code to make network calls (e.g., HTTP over TCP) and to parse the responses. There are several tools and libraries available in the industry that can help to set up and run a mock external web service with less effort, such as Mockoon,[13] Postman,[14] and WireMock.[5] All of them share common functionalities to create configurable mock HTTP-based web services for testing. However, to the best of our knowledge, none of them have the functionality to automatically generate such mock web services.

Creating handwritten mock external web services can be tedious and time-consuming [2]. Furthermore, the process becomes convoluted if the third-party interactions are executed utilizing a provided software development kit (SDK) since the knowledge about interactions may not be available unless provided or the SDK is reverse-engineered. The *record-and-replay* approach can be used to reduce the effort. Instead of manually configuring the mock servers, a pass-through proxy will capture every request and response and utilize them to configure the mock server. However, the external web service should be owned by the same developers of the SUT to be able to emulate more use cases (especially error scenarios), apart from the common "happy path" scenarios. When the system scales up in size, using handwritten mock web services is not a scalable approach. Meanwhile, record-and-replay would still be a feasible method.

[13] `https://mockoon.com/`
[14] `https://www.getpostman.com/`

The work done in [8, 28] is perhaps the closest in the literature to what we present in this paper. However, there are some significant differences. First, we do not use a limited, artificial virtual network but rather make actual HTTP calls using external mocked web services (using WireMock). To be able to achieve this, we have to overcome several challenges, especially when dealing with SSL encryption, and when multiple external web services use the same TCP port. Additionally, to enable effective system-level testing, we need to automatically generate mock responses with valid syntax (e.g., in JSON representing valid DTO for the mocked services), regardless of whether any formal schema is present or not (e.g., the work in [28] requires specifying XML Schema Definition (XSD) files manually with the schema of the response messages). Moreover, in [28], the authors evaluated their techniques only on artificial classes. On the other hand, we show that our novel techniques can scale to real-world systems (e.g., including an industrial API).

To the best of our knowledge, there is a lack of techniques that can fully automate the process of automatic generation of mock external web services in the scientific literature. In addition, the topic of fully automated mock generation of external web services for system-level white-box testing has not received enough attention in the research literature, despite its importance in industry (e.g., considering all the existing popular libraries/tools such as WireMock and Postman). Moreover, previous studies, conducted in mocking the networking interfaces of the Java standard library to use as an input space for search-based fuzzers, have shown significant improvement in code coverage [9, 8]. By simulating the network, the researchers were able to increase the code coverage. Additionally, they were able to produce self-contained test cases which can run independently with all the networking dependencies. Regardless, the studies [9, 8] focused on the individual classes under test rather than on a system level. Furthermore, in [8], the authors used *Mock Objects* only to simulate network connections using a virtual network, and not handling any data sent or received.

2.3.3 Grammars In Fuzzing

Grammar-based fuzzing is another software testing technique where highly effective test inputs are generated based on user-specified grammar. With the help of formal grammar specifications, fuzzers can generate and manipulate well-formed input data. In a previous study [43], the authors combined search-based testing with grammar-based fuzzing and demonstrated that combining those leads to significant improvement in branch coverage for JSON-related classes. Furthermore, previous studies [20, 52] indicate that the use of grammar-based fuzzing significantly improves line coverage and fault detection rate, especially with common input formats such as JSON. Grammar specification typically can be user-specified or can be inferred from documentation (e.g., OpenAPI). Another approach is mining such grammars [26].

Existing literature mainly focuses on grammar-based inputs for endpoints of the SUT. In this work, we focus on automatically generating external web services. Therefore, we need to know about the possible JSON response schemas. There is a possibility to infer the schema from the documentation if it is available. However, this may not always be possible. For instance, if the SUT is using an SDK of a third-party vendor, it may not always be possible to find the input format to create the grammar for further usage. To overcome this challenge, we take an entirely different approach compared to the previous literature. In this paper, we use instrumentation for schema extraction rather than relying on grammar-based techniques such as grammar mining or user-provided grammar (detailed in Section 2.5.4). The reason is that, for REST APIs, JSON is the most common format for data exchange, and, on the JVM, there are two specific libraries that are used for parsing JSON data, which is usually mapped into Java/Kotlin classes. This domain knowledge, if exploited correctly, can significantly simplify and improve the accuracy of the automated inference of the data grammars.

2.3.4 EvoMaster

EvoMaster is an open-source tool used for fuzzing web services using search-based techniques with white-box and black-box fuzzing [13, 5, 17]. EvoMaster provides

client-side drivers (i.e., embedded and external) to enable white-box fuzzing and requires "no code modification" on the SUT, as instrumentation is done automatically. The key difference between embedded and external drivers is whether the SUT and EvoMaster run on the same JVM (i.e., embedded) or in separate JVMs (i.e., external). However, the driver requires writing the necessary steps to cover the three main stages of the API's lifecycle, such as start, reset, and stop. This needs to be manually specified by the user, as APIs can be written in different frameworks and libraries, each one with its own idiosyncratic way of performing these steps. Furthermore, EvoMaster features different search algorithms (e.g., MIO [3] extended with adaptive-hypermutation [53]) and fitness functions (e.g., using advanced testability transformations [11, 15] and heuristics based on the SQL commands executed by the SUT [10]) to generate system-level test suites. In this paper, we implement our novel search-based, white-box fuzzing techniques as an extension of EvoMaster, to enable it to generate system-level test suites with automatic mocking of external web services.

2.4 Motivation

In this paper, our goal is to analyze the impact of automatically handling external web service dependencies during white-box fuzzing. For this purpose, we utilize the EMB corpus of web services [16], which has been used as benchmark in several previous studies [30, 50, 47]. We selected all four APIs that are using external web services for various purposes, out of current 29 from the EMB corpus.[8] Those APIs were used in previous studies without automatic mocking of external web services [54]. As an example, the code block shown the Figure 2.1 is taken from the *GitHub* SDK used in one of our APIs (i.e., *catwatch*[15]).

The class `HttpConnector` is responsible for opening a connection to the given URL using `java.net.HttpURLConnection`. The connectivity with the external web service decides the further execution of the program since the program depends on the response from the external web service to execute its goals. In this particular

[15] `https://github.com/zalando-incubator/catwatch`

```
1  public interface HttpConnector {
2      /**
3       * Opens a connection to the given URL.
4       */
5      HttpURLConnection connect(URL url) throws IOException;
6
7      /**
8       * Default implementation that uses {@link URL#openConnection()}.
9       */
10     HttpConnector DEFAULT = new ImpatientHttpConnector(new HttpConnector()
       {
11         public HttpURLConnection connect(URL url) throws IOException {
12             return (HttpURLConnection) url.openConnection();
13         }
14     });
15 }
```

Figure 2.1: The code snippet taken from *catwatch* [15] shows how an external web service connection is established inside the SUT.

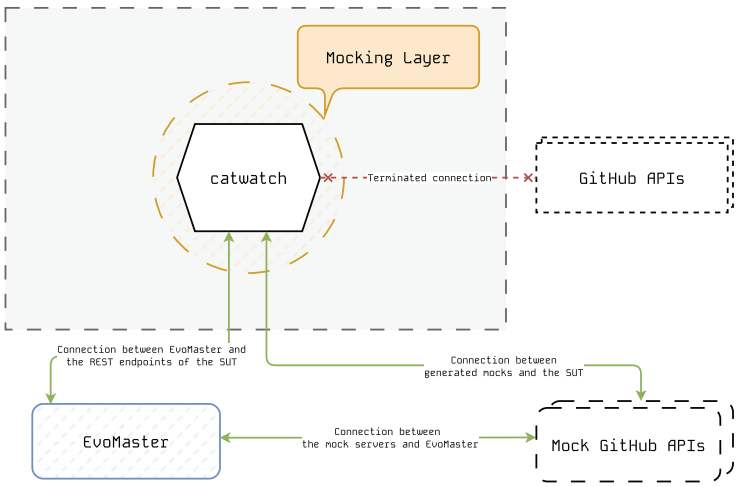


Figure 2.2: The diagram depicts how one of our selected APIs (i.e., *catwatch*) depends on web services to execute its goals. The *Mocking Layer* is the virtual boundary that ends the connection with real external web services.

example shown in Figure 2.1, the *GitHub* SDK will try to connect to *GitHub* web APIs using `HttpURLConnection.openConnection()` to fetch information, as visible in Line 12. For instance, if we manage to mock the particular external web service (e.g., GitHub Web API), we can test the SUT for edge cases that may not be handled inside

the SUT. Figure 2.2 gives a high level representation of how this would work.

Scenarios like this motivated us to build mocking techniques to improve code coverage and fault detection. Since all of our selected APIs are written in Java and Kotlin, we implemented and tested our novel techniques for the JVM ecosystem. Moreover, we implemented our novel techniques as an extension to EvoMaster. Due to this fact, all the decisions made, as discussed in the following sections, favor the compatibility with JVM . Even though we implemented and tested our techniques in the JVM ecosystem, the same techniques can be applied to other languages and runtimes (e.g., *C#*).

2.5 Automated Mock External Services Generation

In this work, we focus on HTTP services that use JSON as data-transfer format, as those are the most common types of web services in industry [41]. To achieve this goal, a good deal of research and technical challenges need to be addressed.

As our novel techniques are designed as an extension to an evolutionary search process, we first start from describing the existing architecture of EvoMaster in Section 2.5.1 and how our novel techniques fit into it. Then, we discuss the details of our novel techniques in five different subsections regarding instrumentation (Section 2.5.2), the mock server (Section 2.5.3), genotype representation (Section 2.5.4), schema inference (Section 2.5.5), and harvesting responses (Section 2.5.6).

2.5.1 EvoMaster Search Process

Figure 2.3 shows a high level overview of the components of EvoMaster, and their relations with the SUT. An EvoMaster *Driver* is responsible to start, stop and reset the SUT. When starting the SUT, the Driver will also instrument the bytecode of the SUT, which needs to be done only once per class in the application. The instrumentation is used to compute different white-box heuristics [4], including advanced testability transformations [12, 15]. The bytecode instrumentation for our novel techniques (Sec-

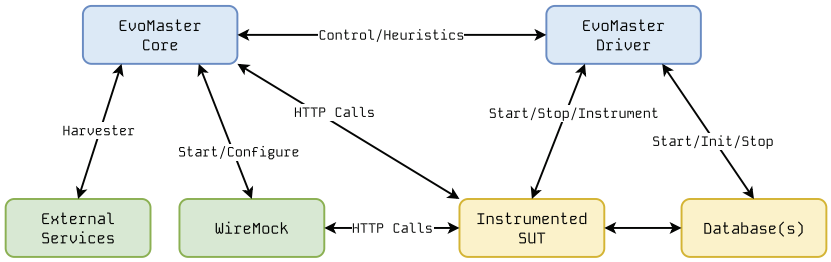


Figure 2.3: High level overview of the components of EvoMaster and their relations to the SUT.

tion 2.5.2) is done here. If the SUT needs any databases (e.g., Postgres, MySQL or MongoDB) up and running, the Driver is responsible to start it (e.g., via Docker). The Driver and SUT can run in the same process (Embedded Driver) or as 2 separated processes (External Driver). The Driver is developed as a REST API, which can be accessed by the *Core* process.

The EvoMaster *Core* process contains the fuzzing engine. It will evolve new test cases, and evaluate them by calling the SUT directly. Then, it will request the Driver to extract the white-box heuristic scores to compute the fitness value of each executed test case.

If the SUT tries to communicate with an external service, such communication will be blocked by our instrumentation. The Driver will then inform the Core of this event. It is the the Core that will start WireMock instances to use instead (Section 2.5.3).

Figure 2.4 shows a simplified overview of the evolutionary search process in EvoMaster, which is executed inside the Core process. Based on the OpenAPI schema of the SUT (which needs to be provided as input to the tool), EvoMaster creates a genotype representation for evolving test cases representing HTTP calls toward the tested REST API. We do not send random bytes on a TPC socket, but rather craft HTTP calls that are valid according the the given schema.

Based on this genotype representation, different search algorithms could be used to evolve test cases. In particular, by default EvoMaster use MIO [3] enhanced with adaptive hypermutation [53]. At a high level, these evolutionary algorithms keep a

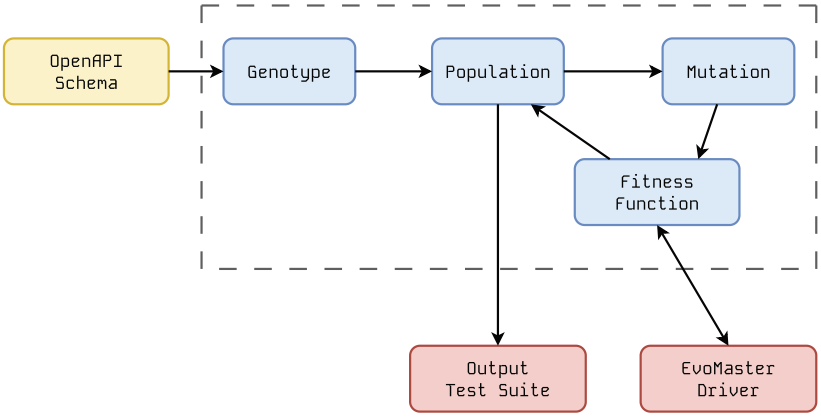


Figure 2.4: Simplified overview of the evolutionary search process in EvoMaster.

population of evolving individuals (e.g., initialized at random, but according to the schema), and then start an evolutionary loop of selecting some of them (based on some fitness score criteria), *mutate* them (e.g., do small random modifications), evaluate their *fitness*, and then decide whether the mutated individuals should be saved back in the evolving populations. This loop process is executed continuously, until the conditions of a stopping criterion are met (e.g., 1 hour has passed). At the end of the search, from the evolved populations a test suite is generated (e.g., in Java using JUnit).

At each fitness evaluation, based on our instrumentation, we can detect each time the SUT tries to interact with an external service. Those interactions are automatically redirected to the WireMock instances started by the Core. Based on which requests are then made by the SUT toward such WireMock instances, the Core will extend the genotype of the evolving test cases to setup responses in the WireMock instances (Section 2.5.4). Based on how the payloads sent by WireMock will be marshalled by the SUT, the genotype is further extended to evolve payloads that will not make the SUT crash when parsing them (Section 2.5.5).

Finally, as shown in Figure 2.3, the Core can make direct calls toward the external services to collect possible payload examples to speed-up the search (Section 2.5.6).

2.5.2 Instrumentation

During the search, we need to automatically detect the hostnames/IP addresses and ports of all the external services the SUT communicates with. To achieve this, we applied a form of *testability transformation* [27], relying on the infrastructure of EvoMaster to apply *method replacements* for common library methods [11] (e.g., the APIs of JDK itself). We extended the white-box instrumentation of EvoMaster with new method replacements for networking APIs. When classes are loaded into the JVM, method calls towards those APIs are replaced with our own developed methods. Those can track what inputs were used, and then call the original API without altering the semantics of the SUT.

More specifically, we create method replacements for Java network classes, such as `java.net.InetAddress`, `java.net.Socket`, and `java.net.URL`. This enables us to collect and manipulate various information related to the connection such as hostname, protocol, and port from the different layers of the Open Systems Interconnection (OSI) model [60]. This approach enables us to capture and analyze the HTTP requests made to the external web service with all their parameters (e.g., URL path, query parameters and headers). Additionally, this allows us to reroute calls made to an external web service to our mock server.

From the first call made to the external web service, we can capture the external web service information through instrumented networking classes such as `java.net.InetAddress`, `java.net.Socket`, and `java.net.URL`. However, in `java.net.InetAddress` case, we are limited with information about the host address. In such event, we rely on other instrumentation such as `Socket` to collect the complete information (e.g., including port number) about the external web services on the subsequent steps.

Meanwhile, we encountered SSL certificate-related issues when dealing with external web service connections done using HTTPS. Typically, when an HTTP client establishes an SSL connection, the client will validate the server using the PKIX algorithm (Public Key Infrastructure X.509). This operation is usually handled by the Java

Secure Socket Extension (JSSE) framework, which is part of the Java Runtime Environment (JRE). If the certificate is invalid, the connection will not be established. As we are running mock servers within the *loopback* address range, it was necessary to find a way to circumvent this verification. In order to maintain the zero-code modification requirement, we are not able to make any modifications to the SUT to utilize our own SSL certificate. Moreover, using our own certificate will make things much more complex. Therefore, we tackle this challenge utilizing instrumentation. For this purpose, we implemented a class to facilitate a custom `X509TrustManager` and `HostnameVerifier` to allow all hostnames during the certificate checks.

Additionally, we created method replacements for various third-party classes from popular networking libraries, such as `okhttp3.OkHttpClient` and `com.squareup.okhttp.OkHttpClient`[16] to use our implementation of `X509TrustManager` and `HostnameVerifier`. This covers the most common cases of dealing with HTTPS calls on the JVM. However, technically there could be other less popular HTTPS libraries in use by the SUT. In these cases, our techniques would not work, albeit supporting such libraries would be just technical work. Arguably, though, in our current implementation we cover the majority of the cases.

Furthermore, information gathered from the other instrumented classes such as `java.net.URL`, `com.fasterxml.jackson.databind.ObjectMapper`, and `com.google.gson.Gson` helped to gather information about the HTTP requests made throughout the search. The collected information is used throughout the search to improve the mock generation. Mock generation utilizing the retrieved information is detailed in Section 2.5.4.

To illustrate, let us consider the example in Figure 2.5. This is a small artificial example of a REST endpoint written in SpringBoot[17] making an HTTP call toward an external service using a `URL` object. We utilized a reserved top-level domain name (i.e., `example.test`), as outlined in RFC 2606, as a dummy external web service in this example.[18] If the request is successful with the response `"HELLO THERE!!!"`, the

[16] `https://square.github.io/okhttp/`
[17] `https://spring.io/projects/spring-boot`
[18] `https://www.rfc-editor.org/rfc/rfc2606.html`

```
1 @GetMapping(path = ["/string"])
2 fun getString() : ResponseEntity<String> {
3
4     val url = URL("http://example.test:8123/api/string")
5     val connection = url.openConnection()
6     connection.setRequestProperty("accept", "application/json")
7     val data = connection.getInputStream().bufferedReader().use(
      BufferedReader::readText)
8
9     return if (data == "\"HELLO THERE!!!\""){
10        ResponseEntity.ok("YES")
11    } else{
12        ResponseEntity.ok("NOPE")
13    }
14 }
```

Figure 2.5: A small example of a REST endpoint written in SpringBoot with Kotlin, making an HTTP call towards an external service using a `URL` object.

```
1 public URLConnection openConnection() throws java.io.IOException {
2     return handler.openConnection(this);
3 }
```

Figure 2.6: Code block from `java.net.URL` which is responsible for opening a connection.

application will respond with an HTTP 200 status code with the string body `"YES"`, otherwise with a `"NOPE"`.

As shown in Figure 2.5, when the `openConnection()` call is executed at Line 5, a connection will be established with the remote destination (at the fictional `example.test:8123`). Once the connection is established, by invoking `getInputStream()` method from `java.net.URLConnection`, we can capture and read the response (Line 7).

The code block shown in Figure 2.6 contains a part of the implementation of `java.net.URL` that is used as the HTTP client library to make a connection. The `handler` in Line 2, represents `URLStreamHandler`. Depending on the protocol (e.g., HTTP, File and FTP), `openConnection` will invoke the respective implementation.

Since we only focus on HTTP-based connections, our instrumentation will replace the call `url.openConnection()` with our custom `URLClassReplacement.openConnection(url)`. Inside this function, we can get all the information about the

HTTP connection, and create a new connection which will connect to a mock server instead of connecting to `example.test:8123`.

Note that this kind of instrumentation is applied to all classes loaded in the JVM, and not just in the business logic of the SUT (i.e., even to third-party libraries used in SUT). For example, if the SUT is using a client library to connect to the external services, this approach will still work. As an example, one of our selected APIs (all APIs used in our empirical study are detailed in Section 2.7), namely *catwatch*, uses the Java library `org.kohsuke:github-api`, which internally connects to `https://api.github.com`.

Meanwhile, during testing, we want to avoid messing up with the connections with controlled services, such as databases. For this reason, we do not apply any modification when the SUT connects to services running on the same host (e.g., *localhost*). This is not a problem when dealing with external services running on the internet (e.g., `https://api.github.com`) or outside of the of SUT host. However, there is a possibility that the external services could run on the same host using different ports. For example, multiple services running in *localhost* using different ports. In general, there is a possibility to change the endpoint address through the SUT configuration. We faced this issue in one of our APIs used as case study mentioned in Section 2.7 (i.e., *cwa-verification*). In the case of *cwa-verification*, this was easily achieved by manually overriding the configuration option `-cwa-testresult-server.url`.

The code block shown in Figure 2.7 is an example implementation of the instrumentation. Through instrumentation, we can control the connection made to external web services and collect various heuristics (e.g., JSON schemas) to improve the search. At the same time, throughout the search, we ensure that method replacements preserve the semantics of the actual methods. For example, in `InetAddress` method replacement, if we are unable to resolve the remote host, an `UnknownHostException` will be thrown like in the real class.

Figure 2.8 depicts how the decision is made when the SUT attempts to establish a connection using `Socket` to an external web service. To establish a connection using `Socket`, it is required to provide the `SocketAddress`. Such address can be presented

```
@Replacement(
        type = ReplacementType.TRACKER,
        category = ReplacementCategory.NET,
        id = "URL_openConnection_Replacement",
        replacingStatic = false,
        usageFilter = UsageFilter.ANY
)
public static URLConnection openConnection(URL caller) throws java.io.
    IOException {
    Objects.requireNonNull(caller);

    URL newURL = ExternalServiceUtils.getReplacedURL(caller);

    return newURL.openConnection();
}
```

Figure 2.7: Implementation snippet of our custom method replacement implementation for the JDK method `openConnection(URL caller)` in the `URLConnection` class.

using an `InetSocketAddress`. In `InetSocketAddress`, it is possible to directly use the remote hostname information, otherwise to use `InetAddress`. Regardless of how an address is represented at the *Socket* level, on the bottom layer, Java uses `InetAddress` to perform the hostname resolution. Due to this, we created method replacements for both `InetAddress` and `Socket`. Figure 2.9 has the sequence diagram of establishing a connection to an external web service using `Socket` while using `InetAddress` to represent the remote hostname in `InetSocketAddress`.

In *Socket*, we can collect information about the protocol, hostname, and port of the external web service. Using such information, we can make a decision about how the connection should be established based on the state of the search as shown in Figure 2.8. Meanwhile, through the `InetAddress` method replacement, we can only collect information about the hostname. Since the protocol and port information are missing, this limits us from initiating a mock server for the respective external web service at this stage. Once the protocol and port information are collected through other method replacements (i.e., *Socket*), a mock server will be initiated in the subsequent stages of the search.

As shown in Figures 2.8 and 2.9, in the *collect heuristics* phase, information about the external web service such as hostname, protocol, and port will be collected through

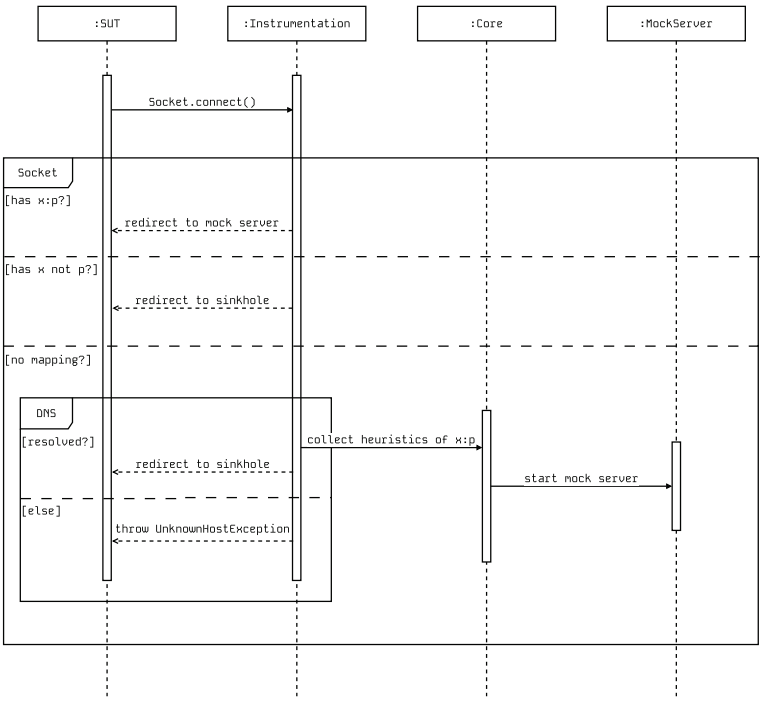


Figure 2.8: Sequence diagram explaining the flow of socket connection between SUT and WireMock. The remote hostname is represented with x and port is represented with p.

method replacements. Apart from the Java networking classes, we created method replacements for third-party libraries such as `OkHttp`,[16] since they use their mechanisms to establish a connection.

We handle the collected information related to DNS (e.g., remote hostname and resolution status) separately from the information related to external web services (e.g., protocol, hostname, and port). We collect information regarding the remote hostname and the resolution status and store it using `HostnameResolutionInfo` for further use throughout the search. Meanwhile, complete information about the external web service such as protocol, hostname, and port will be stored using `ExternalServiceInfo`. Furthermore, this will be used to initiate a mock server to a respective external web service.

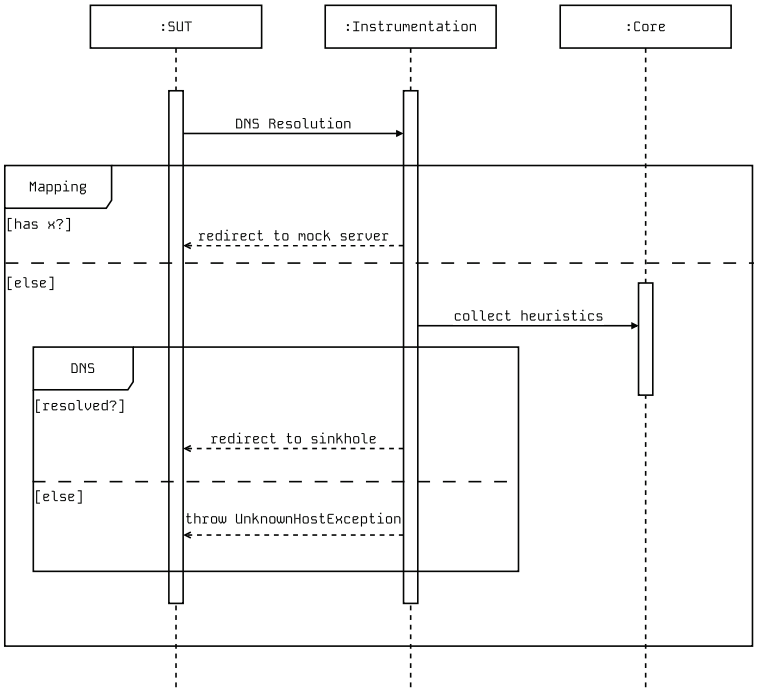


Figure 2.9: Sequence diagram explaining the flow of Domain name resolution when a connection is initiated by the SUT to an external web service. The remote hostname is represented with x.

Additionally, we use a network *sinkhole* to prevent the SUT from connecting to the actual service during the search. A network *sinkhole* is a security technique that is used to redirect network traffic to a controlled environment for analysis or mitigation purposes. In EvoMaster, we allocated an address from the `loopback` (i.e., `127.0.0.2`) range to act as a *sinkhole*, in which the user should ensure no services are running (this is the typical common case, as loopback addresses besides `127.0.0.1` are seldom used). The *sinkhole* address will be used to prevent the SUT from connecting to the actual service. As shown in Figure 2.8 and Figure 2.9, if the external web service is resolved, when the connection is being established, it will be redirected to the *sinkhole*. Otherwise, `UnknownHostException` will be thrown. Additionally, this allows testing the case where the external web service is inaccessible.

Figure 2.7 contains a code block taken from one of the implemented method replacements. `ExternalServiceUtils.getReplacedURL()` method collects the complete information about external web services inside the *URL* and inside the `OkUrlFactory` from `OkHttp`. Apart from that, this method redirects the connection depending on the state of the search, either to the *mock server* or to the *sinkhole*. We use a similar approach in *Socket* to manage the connection being made. However, in the `InetAddress` method replacement, we only replace the actual external web service address (i.e., hostname) with a *mock server* address or with the *sinkhole* address.

When a `HostnameResolutionInfo` is captured through the EvoMaster instrumentation, the `HttpWsExternalServiceHandler` will attempt to allocate a new IP address from the available loopback range, if it is already not allocated. The allocated address will later be used to run a mock server for the respective external web service. Additionally, it will create a mapping between the *remote hostname* and *local IP address*, which will be used inside the instrumented SUT for redirecting the connection to the mock servers. As the next step, the mutator will use the available `HostnameResolutionInfo` to create `HostnameResolutionAction` respectively and will add to the individual as an initialization action. This is to enable the testing of cases in which a hostname does not resolve. Once the search is completed, `HostnameResolutionAction` will be used to set up DNS cache information using `DnsCacheManipulator` in the generated tests.

2.5.3 The Mock Server

Each time an `ExternalServiceInfo` is registered, the `HttpWsExternalServiceHandler` will check for local address mapping and an active mock server. If there is no active mock server available for the respective `ExternalServiceInfo`, `HttpWsExternalServiceHandler` will initiate one.

Writing a mock web server capable of handling HTTP requests is a major engineering effort. Since the focus of this work is not to develop such a tool, we decided to use an existing mock server library. As the implementation of our techniques is targeting

the JVM (e.g., for programs written in Java and Kotlin), we need a mock server capable of running inside the JVM and be able to be programmatically configurable. We chose WireMock[5] for this purpose as it aligns well with our requirements.

WireMock can be configured manually (i.e., by writing all the responses as configurations) or programmatically (i.e., during runtime). Apart from that, WireMock supports record-and-replay as well to configure the mock server. By reconfiguring the software to connect to the WireMock proxy server instead of the target destination, it is possible to capture the requests and responses. Hereafter, captured requests and responses will be remapped as stubs, so the mock server can mimic the behavior of the actual server. Nonetheless, using this approach replicates cases in which the external service works as expected unless unanticipated behaviors are simulated during the recording.

For our purpose, we use WireMock programmatically to handle external web service calls. WireMock allows us to create and run HTTP/S mock servers, and to easily imitate external web services. However, the first challenge is how to instruct the SUT to communicate with these WireMock instances instead of the actual external services. This may not be readily configurable, particularly if the network communications are performed within a client library for the external web service. We addressed this issue by white-box *instrumentation*, as detailed in Section 2.5.2.

Redirecting the traffic to the mock server is not always a trivial task. We tackle this challenge by instrumenting the necessary classes used by the SUT inside the JVM. This instrumentation allowed us to reroute the external web service requests to the respective WireMock server (recall the example in Figure 2.2 discussed in Section **??**).

In general, when a WireMock server receives an incoming request, it will check any registered stub to see how it should respond to the request. A stub can be configured using a static URL path (e.g., `/api/v1/`) or a regular expression-based path (e.g., `/api/v1/.*`). WireMock provides support to extend the request pattern matching using other HTTP request parameters such as headers, body payload, HTTP method, query parameters, cookies, basic authentication, and multipart form data. EvoMaster is designed in a way to restart the search each time when there is an exception. At

the same time, the default behavior of WireMock is to throw an exception if a request pattern is not configured which caused problems during the search. To overcome this challenge, we set WireMock to return a response with an HTTP 404 for all requests if a stub is not present.

Once initiated, all the requests will be redirected to their respective WireMock servers using automated instrumentation, to change the mapping from hostnames to IP addresses. After a test case is executed, we can query WireMock to retrieve the captured request patterns. The default behavior of returning a 404 can be modified to return specific payloads (e.g., JSON responses) for those matched requests.

During the initialization of WireMock servers, it was challenging to manage TCP ports since the SUT may connect to various external web services using the same TCP port (e.g., HTTP 80 or HTTPS 443). Conversely, during the search, we have control over the SUT through instrumentation. On the other hand, it was challenging to maintain the same behavior in the generated test cases (e.g., JUnit files) where no bytecode instrumentation is applied. To ensure the same behavior in the generated test cases as the search, we rely on Java Reflection. Especially when hostnames are hard-coded in third-party libraries and cannot be easily modified by the user (e.g., the case of `https://api.github.com` in *catwatch*).

We managed to modify the JVM Domain Name System (DNS) cache through reflection in the generated test cases by using the `dns-cache-manipulator` from Alibaba.[19] This way, in the generated JUnit tests, we can still remap a hostname such as `api.github.com` to a loopback address where a WireMock instance is running. However, one (arguably minor) limitation here is that we cannot handle in this way the cases in which IP addresses are hard-coded without using any hostname (e.g., using `140.82.121.6` instead of `api.github.com`). At any rate, the use of hard-coded IP addresses does not seem to be a common practice.

Additionally, different operating systems (OS) handle TCP ports in different ways, especially regarding available ephemeral TCP ports and the *TCP Time Wait* delays. *TCP Time Wait* defines the amount of time it will take to release a TCP port once it is

[19] `https://github.com/alibaba/java-dns-cache-manipulator`

freed from the previous process. Each operating system has a different default value for the *TCP Time Wait* delay. This is a major issue for empirical research, especially when running several experiments on the same machine in parallel.

Using different addresses from the loopback address range (e.g.,`127.0.0.0/8`) is an easy and elegant solution to overcome the challenges we faced due to the limitation of available ephemeral TCP ports to initiate WireMock servers. This has well over 16 million addresses, which prevents clash issues in all of our experiments. For example, a WireMock server for `example.test:8123` could be started on `127.1.2.3:8123`. However, some operating systems might handle loopback addresses differently. For example, *macOS* does not enable all possible loopback addresses within the `127.0.0.0/8` range by default. So, other addresses from the range should be configured as an alias on the respective network interface (in most cases, it is *lo0*). By contrast, we have not seen this kind of problem when running experiments on *Linux* and *Windows*. Additionally, it eradicated the dependency on *TCP Time Wait* delays. Furthermore, this allowed us to create and manage mock servers with no code modification required from the SUT.

The decision to pick an IP address to bind mock servers is done in an automated manner. In our extension to EvoMaster, we developed three distinct configurable strategies for address selection. This helped to avoid conflicts while running experiments in parallel and to have more control over address allocation (e.g., in the generated test cases).

In order to have the flexibility in IP address selection during the experiments, we have developed four distinct configurations related to external web service handling inside EvoMaster. Selecting *NONE* will disable the external web service handling. This allowed us to disable the external service handling during the experiments. Under each setting, some of the loopback addresses will be skipped for various reasons. This includes addresses `127.0.0.0`, `127.0.0.1`, `127.0.0.2` (i.e., which is used as `DNS sinkhole`) and `127.0.0.3` which is kept as a placeholder for future purposes. For instance, `127.0.0.1` is skipped due to the limited number of ephemeral ports available, because it is used for various purposes in the operating system as well as in

EvoMaster. Also, we skip the broadcast address of the range `127.255.255.255`, to avoid unanticipated side effects. The *DEFAULT* option will enable the external web service handling and it uses IP addresses starting from `127.0.0.4`. The IP address for the second external web service will be the next in the range (i.e., `127.0.0.5`). Like the name, option *RANDOM* selects a random IP address from the range while excluding the reserved IP addresses. The option *USER* allows the user to specify the starting IP address.

2.5.4 Genotype Representation

As previously mentioned, the instantiated WireMocks will run with a default response of HTTP 404 for all the request patterns. However, mocking the external web services with different correct responses is necessary to improve the code coverage. Recall the example in Figure 2.5, where the result of an `if` statement depends on whether the external service returns the string `"HELLO THERE!!!"`. To reach this point of execution, the mock server should give the response as expected. It would be extremely unlikely to get the right data (i.e., relevant response schema) at random. Performance improvements of the search requires better heuristics in a case like this.

Besides body payloads, the SUT can check other HTTP response parameters as well (e.g., HTTP status code and headers). For instance, the SUT could execute a different code block when the status value is 429 (which denotes too many requests sent within an amount of time). To maximize the code coverage of the SUT (and so indirectly increase the chances of detecting faults), there is the need to have various HTTP responses in each stage of the search to cover all these possible cases.

All of these factors add to the complexity. To tackle these issues, we took a divide-and-conquer approach. All the requests made to external web services are handled individually. Each one of them is represented by an executable action (e.g., `ApiExternalServiceAction`) during the search. Meanwhile, WireMock supports only HTTP(S) based mocking. Consequently, we only tackle HTTP(S) based requests. A REST call may have other functions inside the code, such as calling an external web

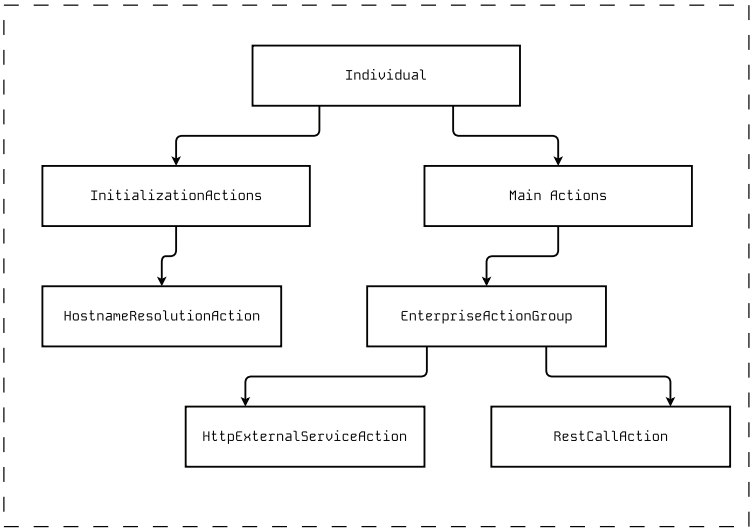


Figure 2.10: The depiction of how actions are divided into groups within each individual.

service, publishing to a message queue, interacting with the database, or reading and writing files. In this paper, we have solely addressed the specifics of how our HTTP(S)-based requests are handled in our methodology.

As the first step, we need to extend the internal fuzzing engine of EvoMaster to deal with the creation of mocked responses. In the evolutionary process of EvoMaster, an evolved *individual* (i.e., a test case) is composed of one or more *actions* (e.g., HTTP calls and data setups in SQL databases). Each action is then composed of a set of *genes*, representing the different parameters defining the action (e.g., string values for URL query parameters and JSON objects for body payloads). Mutator operators of the evolutionary process can alter the structure of an individual by adding or removing actions in it (e.g., adding a new HTTP call in a test case), or modifying the genotype of the actions (e.g., mutating the string value of a query parameter).

Each of the interactions with external services inside the SUT will have a corresponding action inside EvoMaster, to represent how to set the response from the contacted WireMock server. As shown in Figure 2.10, each of the interactions will

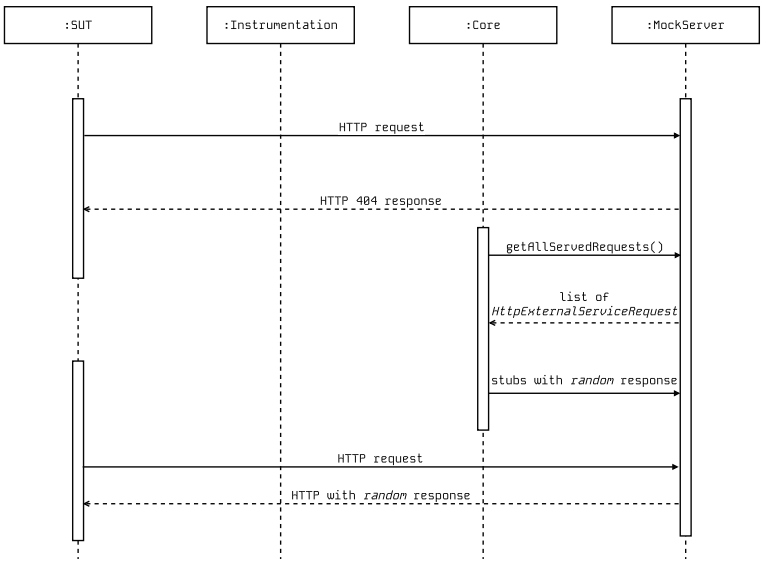


Figure 2.11: The sequence diagram which illustrates how a mock server is set up.

be represented by a child of `ApiExternalServiceAction`, and depending on the protocol, it will have its child action type. In this case, it will be represented using `HttpExternalServiceAction`. All the sub-actions are grouped into an `EnterpriseActionGroup` which will be a child element to the corresponding parent of a `RestCallAction` which defines the API call. `EnterpriseActionGroup` includes all the requests made to the mock server and their respective responses as `HttpExternalServiceAction`. During mutation, if a `RestCallAction` is deleted, all the relevant `HttpExternalServiceAction` inside the `EnterpriseActionGroup` will be deleted along with that.

During the search, once a test is executed, and its fitness evaluation is computed, through the instrumentation we gather information about the interactions the SUT had with the external services from the respective WireMock servers. As shown in Figure 2.11, after `getAllServedRequests()`, the collected heuristics from the mock server, about the served HTTP requests will be used to create `HttpExternalServiceAction` inside the `EnterpriseActionGroup` for each unique request. Each

```
1 val text = BufferedReader(connection.getInputStream().reader()).readText()
2 val tree = gson.fromJson(text, Map::class.java)
```

Figure 2.12: REST endpoint code block is taken from our artificial case study to illustrate taint analysis flow.

HTTP request will be identified as unique by the URL path when creating `HttpExternalServiceAction` and duplicates will be omitted. For example, for `/api/v1/` there will be only one `HttpExternalServiceAction` representing the request, regardless of how many requests are made to the external web service.

At the point of initiation, an instance of `HttpExternalServiceAction` will have information about the request made to the external web service such as hostname, protocol, URL path, headers, and body. In the following steps of the search, when the test is selected again for mutation, these newly created actions will be mutated. Therefore, the respective WireMock server will have a new response for the request pattern besides the default `HTTP 404`. Initially, the genotype of the `HttpExternalServiceAction` will contain mutable genes to handle the returned HTTP status code and an optional body payload (for example, treated as random strings). As a result, a new mock server response will be made using randomly picked predefined values of HTTP status in the `HttpWsResponseParam`. The response body will be decided based on the selected HTTP status code. For HTTP status codes such as all 1xx (Informational), 204 (No Content), and 304 (Not Modified) responses do not include content. For other HTTP status codes, the response body will be an empty string at the initiation. Similarly, throughout the search, actions will enhance the corresponding WireMock stubs by mutating those genes and then collecting their impact on the fitness function.

2.5.5 Schema Inference

It is possible to easily randomize some parameters (e.g., HTTP status codes) in a typical HTTP response during the search. However, creating correct values in parameters like headers and response bodies is not a straightforward task. With no information available at the beginning of the search, it is necessary to find a way to gather this

information to create a response schema. For example, when the SUT expects a specific JSON schema as a response, sending a random string as the body payload will lead to throwing an exception in the data parsing library of the SUT.

On the other hand, the information about the external web service may be available in a commonly used documentation format such as OpenAPI/Swagger. In theory such information could be used to create syntactically valid responses. However, not only such formal schemas could be missing, but also automatically locating them might not always be possible unless the user provides them manually. As an alternative, more general solution would be to use bytecode instrumentation to analyze how such body payloads are parsed by the SUT [15]. Furthermore, this approach allows us to extract the essential schema information needed to generate the mock web services.

The JSON is a common choice as a data transfer format for the interactions between web services [41]. Usage of Data Transfer Objects (DTO) is a known practice in most of the statically typed programming languages (e.g., Java, C, C++, Rust) to represent the schema in code. When a JSON payload is received from an HTTP request, a library can be used to parse such text data into a DTO object. By instrumenting the relevant libraries at runtime during the test evaluation, we can analyze these DTOs to infer the schema structure of these JSON messages.

On the JVM, the most popular libraries for JSON parsing are *Gson* and *Jackson* [33]. For these two libraries, we provide method replacements for their main entry points related to parsing DTOs, for example, `<T> T fromJson(String json, Class<T> classOfT)`. In these method replacements, we can see how any JSON string data is parsed to DTOs (analyzing as well as all the different Java annotations in these libraries used to customize the parsing, for example, `@Ignored` and `@JsonProperty`). This information can then be fed back to the search engine: instead of evolving random strings, EvoMaster can then evolve JSON objects matching those DTO structures.

There is one further challenge that needs to be addressed here: how to trace a specific JSON text input to the source it comes from. It can come from an external web service we are mocking, but that could use different DTO for each of its different endpoints. Alternatively, the data may come from a database or may be provided as input

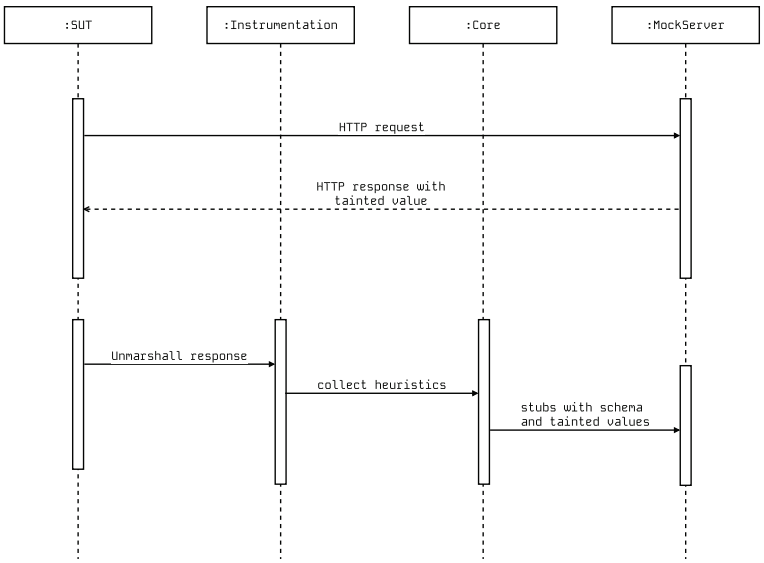


Figure 2.13: The sequence diagram demonstrating the flow of taint analysis.

to the SUT. In those cases, the data has nothing to do with WireMock instances. Therefore, we need to distinguish each case.

In order to tackle such a challenge, we use *taint analysis*. Figure 2.13 explains the flow of how the taint analysis is implemented to achieve this goal. In particular, we extend the current input tracking system in EvoMaster [12]. Each time a string input is used as body payload in the mocked responses, it will not be a random string but rather a specific tainted value matching the regex `_EM_\d+_XYZ_`, e.g., `_EM_0_XYZ_`.

Figure 2.12 contains two lines of code obtained from one of our artificial case studies to provide a practical example of taint analysis. In Line 2, the SUT reads the response from the external service call made using `BufferReader`. Afterwards, using `fromJson` from *Gson* library, the response JSON string is converted into a `Map` of strings to process further. In this case, during the input generation, by including a tainted value (e.g., `_EM_0_XYZ_`) in the JSON string it is plausible to trace the process of input handling throughout the execution. Such a mechanism allows us to trace and identify the DTO used. In the next step, the identified DTO can be used further as the response

schema in the mock server for the request.

Similar to the provided example in Figure 2.12, in each of our method replacements (e.g., for `fromJson`), we check if the input string matches that regular expression. If so, we can check the genotype of the executed test case to see where that value is coming from. The gene representing that value is then modified in the next mutation operation to rather represent a JSON object valid for that DTO.

During the search, modifications/mutations to an evolving test case might lead the SUT to connect to new web services, or not connect anymore to any (e.g., if a mutation leads the test case to fail input validation, and then the SUT directly returns a 400 status code without executing any business logic). During the fitness evaluation all the mock servers will be reset to the default state before computing the fitness (i.e., with the HTTP 404 for all request patterns). Thereafter, the active state of the action will be set based on the "used" parameter for all existing actions. Active actions will rebuild the WireMock stubs based on the active index for each HTTP parameter. After the test execution, if no matching request exists to the existing actions after a fitness evaluation, it will be marked as not used (i.e., "used" will be set to false). The rest will be marked as used for further use.

2.5.6 Harvesting Responses

Inferring the expected structure of the JSON responses can improve the search performance (as we will show in Section 2.7). However, it might not always be possible to infer these schemas. Furthermore, even when possible, for the search to generate a useful response to mock the external web service, it might take a significant amount of time.

To expedite this process, we have implemented an approach to harvest the actual responses from the original service in parallel to the search, if it is available. This means that, when the SUT makes a call to a WireMock server, from our EvoMaster extension, we still do the same call as well towards the real external web service. We created a multithreaded harvester to make these requests and fetch the responses with all their

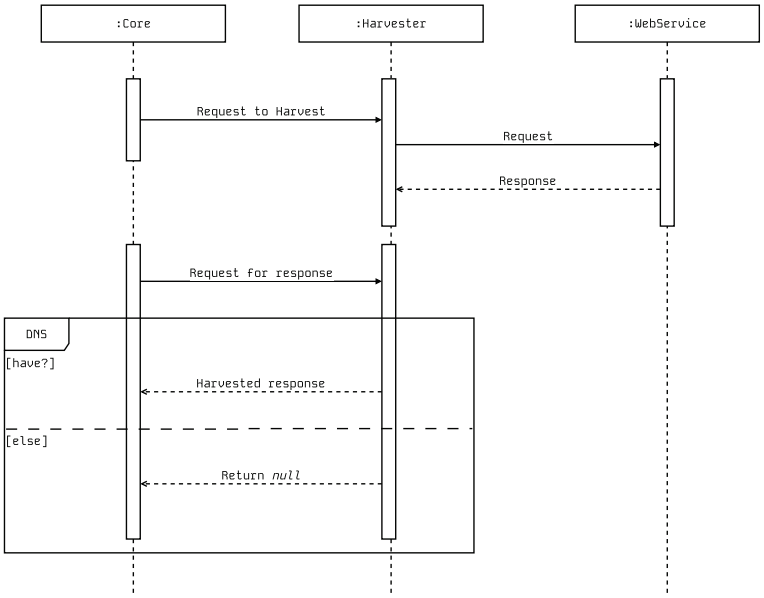


Figure 2.14: Sequence diagram explaining the execution flow of harvesting responses from the actual external web services.

relevant information, such as headers. Then, each time a test case is mutated, it will refer to the harvester for available actual response payloads to set up the WireMock stubs, as illustrated in Figure 2.14. With a given probability, one of these responses (for the matching called endpoint) will replace the current genotype of the WireMock action setup. Then, these payloads can still be further mutated and evolved during the search.

Anyhow, accessing the actual service is not always possible (e.g., for the *cwa-verification* case the service was not publicly available) and accessible. Sporadically, we faced issues with the availability of the service, and sometimes rate-limited access. Apart from this, occasional response timeouts and network performance degradation caused issues. This had an impact on the search performance.

To overcome these hurdles, we developed three strategies to select the harvested actual responses, if available. The first option selects the exact matching response to the request URL, the second option selects the closest match to the request URL, and

the third option selects a random response from the harvested requests. The selection is a continuing process until the search finds a satisfying response to the request. We evaluated each option's performance separately. In some preliminary experiments (not reported in this paper), we did not see much differences among these strategies, so we left on the default random strategy.

2.6 Generated Test Cases

During the search, we were able to control the external web service connections using instrumentation. Instrumentation helped to manipulate the URL information to connect the HTTP mock servers instead of the external web services. Unfortunately, replicating the same behavior within the generated JUnit test cases has become another challenge. To do the same inside the generated test cases, we have to rely on the DNS cache of the JVM.

The JVM uses DNS caching to improve the network performance. Each first network access to a remote location made by the application will cache the DNS information for further use. So, any subsequent operation will read from the previously cached information. The cached DNS information inside the JVM can be programmatically altered if it is necessary to reroute the traffic to a different destination for the same hostname.

In our case, before each call to external web services, we are able to manipulate the DNS cache information using a library called *DNS cache manipulator*. *DNS cache manipulator* helped to reduce the amount of work that needs to be done to achieve the same results [19].

At the end of the search, EvoMaster produces self-contained test suites as JUnit test files, written in either Java or Kotlin. An individual test suite will contain all the necessary configurations related to WireMock, and this allows the test to run independently without any additional configurations. Figure 2.15 has a single test taken from the generated test suite for the synthetic example in Figure 2.5. After the function definition, `DnsCacheManipulator.setDnsCache()` configures the JVM DNS

```
@Test @Timeout(60)
fun test_7()  {
    DnsCacheManipulator.setDnsCache("example.test", "127.0.0.92")
    assertNotNull(wireMock__http__hello_there__8123)
    wireMock__http__hello_there__8123.stubFor(
        get(urlEqualTo("/api/string"))
        .atPriority(1)
        .willReturn(
            aResponse()
                .withHeader("Connection","close")
                .withHeader("Content-Type","application/json")
                .withStatus(201)
                .withBody("\"HELLO THERE!!!\"")
            )
    )

    given().accept("*/*")
            .header("x-EMextraHeader123", "")
            .get("${baseUrlOfSut}/api/wm/urlopen/string")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("text/plain")
            .body(containsString("YES"))


    wireMock__http__hello_there__8123.resetAll()
}
```

Figure 2.15: The code block taken from a generated test specified with JUnit and Kotlin for the synthetic example shown in Figure 2.5.

cache to redirect all the DNS queries to a local address where the WireMock server is running. With the successful assertion on the presence of WireMock, the test further sets the stub with all the necessary parameters for the request pattern for a successful response. This stub represents the response from the external web service with which the SUT is supposed to communicate. Further down, the test checks for the successful response from the SUT. This is only achievable when the external web service responds with a `"HELLO THERE!!!"` text, as shown in Figure 2.5.

Table 2.1: Experiment settings

RQ	Internet	Techniques	SUTs
RQ1	On	*Mocking*	Artificial
RQ2	On	*Base*, *Mocking*	Real-World
RQ3	Off	*Base*, *Mocking*	Real-World

2.7 Empirical Study

To evaluate the effectiveness of our novel techniques presented in this paper, we carried out an empirical study to answer the following research questions:

RQ1: Is our novel approach capable of automatically generating mock external web services for artificial, hand-crafted examples?

RQ2: Compared to the performance of existing white-box fuzzing techniques, how much improvement can our mock external web service generation achieve in terms of code coverage and fault detection on real-world applications?

RQ3: When external services are inaccessible, how effective is our approach in generating mock external web services?

2.7.1 Experiment Setup

To answer our three research questions, we carried out three different sets of experiments, one per research question, on two different sets of SUTs, as shown in Table 2.1. We consider two different variables: (1) the *internet* (either *On*, or *Off*) represents whether the internet connection is enabled or not while running the experiments; (2) then we conducted experiments to explore that the *technique* represents the baseline and our proposed approach, i.e., *Base* refers to EvoMaster with its default configuration, and *Mocking* refers to our approach, which enables our handling of mock external web service generation with EvoMaster.

Table 2.2: Descriptive statistics of the employed artificial SUTs. For each SUT, #Endpoints represents the number of declared HTTP endpoints in those APIs. #Path represents the source code path in the E2E test folder of EvoMaster.[9]

SUT	#Endpoints	#Path
Auth0	1	com/foo/rest/examples/spring/openapi/v3/wiremock/auth0
INet	1	com/foo/rest/examples/spring/openapi/v3/wiremock/hostnameaction
Socket	3	com/foo/rest/examples/spring/openapi/v3/wiremock/socket
URL	3	com/foo/rest/examples/spring/openapi/v3/wiremock/urlopen
OkHttp	3	com/foo/rest/examples/spring/openapi/v3/wiremock/okhttp
OkHttp3	3	com/foo/rest/examples/spring/openapi/v3/wiremock/okhttp3
Harvester	4	com/foo/rest/examples/spring/openapi/v3/wiremock/harvestresponse

Artificial APIs

To answer **RQ1**, the first set of SUTs consists of seven artificial APIs, developed by us, as listed in Table 2.2. This includes REST APIs covering different libraries and various real-world scenarios. The list includes third-party libraries such as `Auth0`[20] and `OkHttp`, as well as JDK classes such as `InetAddress`, `Socket`, and `URL`. Additionally, this set contains as well an API to cover the *Harvester* (i.e., covering the technique presented in Section 2.5.6) using a few real-world APIs as external web services. Besides, we use a popular JSON parser, namely *Jackson*,[7] for JSON handling inside the artificial APIs. The artificial examples were developed using SpringBoot[17] with Kotlin.

Fuzzing a REST API involves addressing several different challenges [54]. Our novel techniques presented in this paper are aimed at one specific challenge, which is dealing with interactions with external services. To obtain sound empirical evidence, empirical evaluations should be based on real-world applications. However, lacks of performance improvements on some APIs might be independent of the quality and effectiveness of a novel technique, and rather they could be related to other challenges that are not fully solved yet. For example, no communications with external services would be done if a fuzzer is not able to generate test cases that can bypass the first layer of input validation, and so the SUT would return immediately with a 4xx user-error status code. Such an API would not be good for studying the effectiveness of mocking

[20] `https://auth0.com/`

```
@GetMapping(path = [""])
fun get() : ResponseEntity<String> {

  try {
    val domain = "www.doesnotexistfoo.test:6789"
    val audience = String.format("https://%s/api/v2/", domain)
    val authClient = AuthAPI(domain, "foo", "123")

    val tokenHolder = authClient.requestToken(audience).execute()
    return ResponseEntity.ok("OK")
  } catch (e: Exception){
    return ResponseEntity.status(400).build()
  }
}
```

Figure 2.16: REST endpoint code block which is taken from our artificial API *Auth0* using the `Auth0` SDK.

external services, as no call to external services would be done.

To address these potential issues, we have designed artificial examples in which dealing with external services is the main (and possibly only) challenge. This enables us to study the effectiveness of our novel techniques without any confounding factors. Furthermore, this also enables us to have a larger case study, as finding more real-world APIs for experimentation in this context is challenging. Still, experiments only on artificial examples are not sufficient. Real-world applications must be used as well.

The full source code of these artificial examples is available in the code repository of EvoMaster, as they are currently used as end-to-end (E2E) tests for it [14]. Table 2.2 provides the package paths for each example. In the reminder of this section, for sake of clarity, we also provide code snippets for some of the endpoints of these artificial APIs.

Auth0. Figure 2.16 contains the REST endpoint written in SpringBoot.[17] As shown, we initiate the `Auth0` SDK with a fake hostname and port. This library will try to connect to that remote server, assuming it is a an Auth0 server, where it will try to authenticate the user called `foo`, with password `123`. If the connection is successful through the SDK, our REST API will respond with an HTTP 200 status code, and with the body `"OK"`. Otherwise, it will answer with a user error HTTP 400 status code.

Inet. Figure 2.17 contains the code of the REST endpoint from the artificial case

```
1 @GetMapping(path = ["/resolve"])
2 fun exp(): ResponseEntity<String> {
3   val address = InetAddress.getByName("imaginary-second.local")
4
5   return try {
6     val socket = Socket()
7
8     val a = InetSocketAddress(address, 10000)
9     socket.connect(a, 10000)
10    socket.close()
11
12    ResponseEntity.ok("OK")
13  } catch (e: Exception) {
14    ResponseEntity.status(400).build()
15  }
16 }
```

Figure 2.17: REST endpoint code block which is taken from our artificial API *Inet* for `java.net.InetAddress`.

study developed to test the usage of `java.net.InetAddress`. First, the code initiates an `InetAddress` with a fake hostname and uses it to create an `InetSocketAddress` (Lines 3– 8). After that, the SUT tries to connect to the remote host using a `Socket` (Line 9). Once the connection is established successfully, the code will respond with HTTP 200 and body `"OK"`, otherwise with HTTP 400. Although this implementation uses `java.net.Socket`, as shown in Figure 2.18, we have created separate APIs to cover scenarios related to the `Socket` class.

Socket. Similar to the *Inet* API, in the *Socket* API the endpoint tries to resolve a fake hostname and port using `java.net.InetSocketAddress` instead of a `java.net.InetAddress`, and then establishes a connection, as shown in Figure 2.18. On success, the endpoint will respond with HTTP 200 and body `"OK"` (Line 12), otherwise with HTTP 400 (Line 14).

URL. Unlike *Inet* and *Socket*, for `java.net.URL` we have three different cases. The code block in Figure 2.19 is taken from the API developed to study `java.net.URL`. Each case uses `java.net.URL` to establish a connection to a fake external web service (Lines 2– 3). We developed two identical endpoints to test HTTP and HTTPS. Additionally, we created one more endpoint to use `com.fasterxml.jackson.databind.ObjectMapper` from *Jackson*.

```
1 @GetMapping(path = ["/resolve"])
2 fun exp(): ResponseEntity<String> {
3   val a = InetSocketAddress("imaginary-socket-evomaster.com", 12345)
4
5   return try {
6
7     val socket = Socket()
8
9     socket.connect(a)
10    socket.close()
11
12    ResponseEntity.ok("OK")
13  } catch (e: Exception) {
14    ResponseEntity.status(400).build()
15  }
16 }
```

Figure 2.18: REST endpoint code block taken from our artificial case study for `java.net.Socket`.

```
1 ...
2 val url = URL("http://example.test:8123/api/string")
3 val connection = url.openConnection()
4 connection.setRequestProperty("accept", "application/json")
5 ...
```

Figure 2.19: A code block taken from the API *URL* developed for `java.net.URL`.

Under each endpoint, once the connection to an external web service is successful, each uses a different means to unmarshal the received response. In two of the endpoints, we used the *URL* built-in buffer reader, while in one we used `com.fasterxml.jackson.databind.ObjectMapper` from *Jackson* to unmarshal the response. In the end, in two of the endpoints, we return `"OK"` with HTTP 200 if the external service responds with `"HELLO THERE!!!"`, otherwise with HTTP 500. Meanwhile, in one that uses the `ObjectMapper`, we return `"OK"` with HTTP 200 if the integer parameter (i.e., x) in the response is greater than 0.

OkHttp and *OkHttp3*. Similar to *URL*, we have three different endpoints for `com.squareup.okhttp` and `okhttp3`. The code snippet in Figure 2.20 is taken from the API created to test the `com.squareup.okhttp`. First, `com.squareup.okhttp.Request` is used to establish a connection to a remote host using different ports and protocols, such as HTTP and HTTPS (Lines 2 and 3). In this example, HTTP protocol is

```
1 ...
2 val url = URL("$protocol://$host:5555/api/string")
3 val request = Request.Builder().url(url).build()
4
5 try {
6   val data = client.newCall(request).execute()
7   val body = data.body()?.string()
8   val code = data.code()
9   return if (code in 200..299){
10    if (body == "\"HELLO THERE!!!\""){
11      ResponseEntity.ok("Hello There")
12    }else{
13      ResponseEntity.ok("OK")
14    }
15  } else if (code in 300..499){
16    ResponseEntity.status(400).build()
17  }else{
18    ResponseEntity.status(418).build()
19  }
20 }catch (e: Exception){
21   return ResponseEntity.status(500).build()
22 }
23 ...
```

Figure 2.20: A code block from the API *OkHttp*.

used for connecting to `github.com` in port 5555. Likewise, we have developed a separate method to use HTTPS as well. Furthermore, similar to the *URL*, we have one more method to test the use of `com.fasterxml.jackson.databind.ObjectMapper` from *Jackson*.

Harvester. To test the our Harvester technique (recall Section 2.5.6), we created a dedicated API with four endpoints to cover different scenarios. Compared with previous examples, in *Harvester* instead of using fake hosts, we used one public API used in *genome-nexus* and an API with a JSON schema list created in *Listly*[21] with synthetic data. Furthermore, in the *Harvester* API, we used all the libraries from previous examples to make connections and to read responses (i.e., *URL*, *Jackson*, and *OkHttp*).

Table 2.3: Descriptive statistics of the employed SUTs. For each SUT, #SourceFiles represents the number of files (i.e., the number of public Java classes) composing such SUT, where #LOCs represents their total number of lines of code (LOC). Finally, #Endpoints represents the number of declared HTTP endpoints in those APIs.

SUT	#SourceFiles	#LOCs	#Endpoints
catwatch	106	9636	14
cwa-verification	47	3955	5
genome-nexus	405	30004	23
ind1	163	15240	53
pay-publicapi	377	34576	10
Total 5	1098	93411	105

Real-World APIs

To answer **RQ2** and **RQ3**, we decided to evaluate our approach using real-world applications. Therefore, our second sets of SUTs consist of four APIs from EMB [16] and one industrial API (i.e., *ind1*). EMB is an open-source corpus composed of open-source web/enterprise APIs for scientific research [16]. This corpus has been utilized by different research groups for performing experiments related to Web API fuzzing techniques [30, 47, 50, 57].

To assess the effectiveness of our mock generation, from EMB, we selected all the REST APIs which require communicating with external web services for their business logic (i.e., *catwatch*, *cwa-verification*, *genome-nexus*, and *pay-publicapi*). Descriptive statistics of the selected APIs are reported in Table 2.3. In total, 105 endpoints and 93*k* lines of codes were employed in these experiments. Note that these statistics only concern code for implementing the business logic of these APIs. The code related to third-party libraries (which can be millions of lines [16]) is not counted here.

Compared to other studies in the literature on fuzzing REST APIs, using only four open-source projects can be considered limited. For example, in the study introducing ARAT-RL [29] 10 open-source APIs were employed, and 11 were employed in the study introducing DeepREST [18]. In our own previous work on comparing REST API

[21] `https://list.ly`

fuzzers, we used 18 open-source APIs [54]. As our novel techniques are white-box based, we are limited in our selection of APIs written in the programming languages we can support, i.e., Java and Kotlin. Furthermore, although intra-API communications are common in industrial APIs, especially when large systems are developed with a microservice architecture, they are less so in open-source projects. We are aware of no other Java/Kotlin API used in previous research studies in the literature that we could have added for our empirical study in this paper. This is a major reason why we made sure to employ in our study at least one closed-source industrial API, and developed seven artificial APIs.

catwatch is a web application that retrieves statistics from GitHub for user accounts, processes and stores this data in a database, and subsequently provides access to the data through a REST API. The application heavily relies on GitHub APIs to perform its core functions.

cwa-verification is a component of the official Corona-Warn-App for Germany. This API is part of a microservice-based architecture that contains various other services, including mobile and web apps. The tested backend server relies on other services in the microservice architecture to complete its tasks.

genome-nexus is an interpretation tool that allows gathering information about genetic variants in cancer. This API gathers and integrates information from various external online sources. These include web services that convert DNA changes to protein changes, predict the functional effects of protein mutations, and contain information about mutation frequencies, gene function, variant effects, and clinical actionability.

pay-publicapi is an open-source API from the government of the United Kingdom for government or public sector organizations that want to take payments. The API provides the ability to take payments, issue refunds, and run reports on all payments. This API is currently being used by partners from across the public sector, including the NHS, MOJ, police forces, and local authorities.

ind1 is closed-source API, part of an e-commerce system. It deals with authentication using an external Auth0 server, and it processes monetary transactions using Stripe.[2]

Interactions with real external web services are often non-deterministic, as the external web services might not be accessible and change their behavior at any time. This is a major issue for software testing, as any generated test could become flaky [44]. In this study, to handle this issue, we studied the performances of the compared techniques with two internet settings (i.e., *On* and *Off*). The interactions are deterministic with the disabled internet setting (i.e., *Off*), while, with an enabled setting (i.e., *On*), results might be negatively affected by the variance in the external web services. For instance, with a preliminary study, we found that *catwatch* communicates with GitHub API, but there exists a rate limiter (a typical method to prevent denial-of-service attacks) to control the access to this API[22] based on IP address, e.g., up to 60 requests per hour for unauthenticated requests. As the network setup of our university, we may share the same IP address with all of our colleagues and students (e.g., when behind a NAT router). Due to this, such a rate limit configuration will strongly impact the results of the experiments. For example, depending on the time of the day in which the experiments are running, in some experiments we might be able to fetch data from GitHub, but not in others. As this can lead to completely unreliable results when comparing techniques, we decided to run experiments without internet connection as well. Therefore, to answer **RQ3**, we repeated the same experiments as **RQ2**, but with the machine physically and wirelessly disconnected from the internet. This also provides a way to evaluate our techniques in cases in which the external web services are not implemented yet (e.g., at the beginning of a new project) or are temporarily down.

Considering the random nature of search algorithms, we repeated each setting 30 times using the same termination criterion (i.e., 1 hour), by following common guidelines for assessing randomized techniques in software engineering research [6]. With two settings (i.e., *Base* and *Mocking*) on five SUTs using 1 hour as search budget, 30 repetitions of the experiments took 300 hours (12.5 days), i.e., $2 \times 5 \times 30 \times 1$, for **RQ2**, and another 300 hours for **RQ3**, for a total of 25 days.

All the experiments were run on the same machine, i.e., an HP Z6 G4 Workstation

[22] https://docs.github.com/en/rest/overview/resources-in-the-rest-api?apiVersion=2022-11-28

with Intel(R) Xeon(R) Gold 6240R CPU @2.40GHz 2.39GHz, 192 GB RAM, and 64-bit Windows 10 OS.

To take the randomness of these algorithms into account when analyzing their results, we used common statistical tests recommended in the literature [6]. In particular, we employed a Mann–Whitney U test (i.e., p-value) and Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) to perform comparison between *Base* and *Mocking* in terms of line coverage and fault detection. With the Mann–Whitney U test, if p-value is less than a significant level (i.e., 0.05), it indicates that the two compared groups (i.e., *Base* and *Mocking*) have statistically significant differences. Otherwise, there is not enough evidence to support the claim the difference is significant. Comparing *Mocking* with *Base*, the Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) measures how likely *Mocking* can perform better than *Base*. $\hat{A}_{12} = 0.5$ represents no effect. If $\hat{A}_{12}$ surpasses 0.5, it indicates that *Mocking* has more chances to achieve better results than *Base*.

2.7.2 Results for RQ1

In this section, we discuss the details of the experiments to be able to answer **RQ1**. Due to space constraints, we only discuss the experiment steps for *Auth0* in detail. For the other APIs, the experiment steps were identical.

Each artificial API is designed in a way in which it makes a call to an external service (that does not exist), fetches some data, parses it, and then returns a success response if no error was encountered. If our novel techniques are successful, then it should be possible to get a success response on each of these artificial APIs. We do not need to empirically compare with *Base* technique here, as by construction all test executions will crash when trying to connect to the non-existent external services. As such, we already know it would be impossible to get any success response with *Base*.

In a preliminary study, we successfully generated self-contained test suites with mock external web services for all the artificial APIs presented in Table 2.2. Furthermore, we were able to handle multiple mock external web services at once without any conflicts, as discussed in Section 2.5.3. All these artificial APIs can now be fully covered

```
1 @Test
2 fun testRunEM() {
3   runTestHandlingFlakyAndCompilation(
4     "WmAuth0EM",
5     "org.foo.WmAuth0EM",
6     500,
7     true,
8     { args: MutableList<String> ->
9
10      args.add("--externalServiceIPSelectionStrategy")
11      args.add("USER")
12      args.add("--externalServiceIP")
13      args.add("127.0.0.22")
14      args.add("--instrumentMR_NET")
15      args.add("true")
16
17      val solution = initAndRun(args)
18
19      assertTrue(solution.individuals.size >= 1)
20      assertHasAtLeastOne(solution, HttpVerb.GET, 200, "/api/wm/auth0", "OK
    ")
21    },
22    3
23  )
24 }
```

Figure 2.21: E2E test case (in the class named `WmAuth0EMTest`) of EvoMaster for the *Auth0* API.

by EvoMaster, using a small search budget, reliably. Thanks to this, these artificial APIs have been added as part of the E2E tests for EvoMaster [14]. This means that, at each build of EvoMaster on its Continuous Integration server (currently GitHub Actions), EvoMaster is run on these APIs, and the build fails if any API cannot be covered. To achieve this, we wrote embedded drivers and E2E tests for each of these APIs.

In the following steps, we discuss the details of the tests and the outcomes in detail only for *Auth0*, as an example. The other APIs would have similar discussions.

As shown in Figure 2.21, we wrote an EvoMaster E2E test to evaluate our artificial API with *Auth0* SDK. This test uses the E2E scaffolding of EvoMaster to run the API, run EvoMaster on it (with a search budget of 500 fitness evaluations in this case), verify the defined assertions (e.g., Line 20), compile the generated JUnit test cases, and run those later as well. If the E2E test takes more than 3 minutes (specified on Line 22),

```
1 private val controller : SutHandler = com.foo.rest.examples.spring.openapi.
    v3.wiremock.auth0.WmAuth0Controller()
2 private lateinit var baseUrlOfSut: String
3 private lateinit var wireMock__https__www_doesnotexistfoo_test__6789:
    WireMockServer
```

Figure 2.22: Variable declarations from the generated suite for the SUT (i.e., *Auth0*) shown in Figure 2.16.

it will fail. If the test is flaky, it is repeated up to N times with different incremental seeds. The full details of this E2E test infrastructure can be found in [14].

In this E2E test, we evaluate whether we are able to reach the line in the SUT which gives the response of HTTP 200 and with the body `"OK"`. To be able to reach the line of code that returns the HTTP 200 response, the implementation of our approach of automatically generating external web services must work successfully. Novel, experimental features are not on by default, until they are stable and properly analyzed. In this case, our novel techniques need to be activated via commandline properties, as set in this E2E test on Line 14. If we manage to reach the line of code within the given budget for search, the test will pass (i.e., as the assertion on Line 20 will be satisfied). Finally, the test will generate a self-contained JUnit test suite after completion. Below is a detailed analysis of these generated tests.

As seen in Figure 2.22, the beginning of a test suite contains the necessary variables to configure the mock servers and other parameters needed throughout the test. Of particular importance is the instantiation of the embedded driver (class `WmAuth0Controller` on Line 1). Figure 2.23 shows the `BeforeAll` section of the generated test suite. Under the `BeforeAll` section, the test suite will use the driver to initiate the SUT (Line 5). After the successful start of the SUT, the test suite will set up the mock server(s) and start them (Line 18). Each test is self-contained in the sense that the necessary configuration to run the test successfully will be configured under the test, and everything will be reset to default at the end.

Figure 2.24 contains the `BeforeEach` section of the generated test suite. Before executing each test, we ensure that the mock server is always set to the default state, similar to the behavior during the search, as discussed earlier in Section 2.5.3. Additionally,

```
@BeforeAll
@JvmStatic
fun initClass() {
  controller.setupForGeneratedTest()
  baseUrlOfSut = controller.startSut()
  controller.registerOrExecuteInitSqlCommandsIfNeeded()
  assertNotNull(baseUrlOfSut)
  RestAssured.enableLoggingOfRequestAndResponseIfValidationFails()
  RestAssured.useRelaxedHTTPSValidation()
  RestAssured.urlEncodingEnabled = false
  RestAssured.config = RestAssured.config()
    .jsonConfig(JsonConfig.jsonConfig().numberReturnType(JsonPathConfig.
    NumberReturnType.DOUBLE))
    .redirect(redirectConfig().followRedirects(false))
  wireMock__https__www_doesnotexistfoo_test__6789 = WireMockServer(
    WireMockConfiguration()
    .bindAddress("127.0.0.22")
    .httpsPort(6789)
    .extensions(ResponseTemplateTransformer(false)))
  wireMock__https__www_doesnotexistfoo_test__6789.start()
}
```

Figure 2.23: `BeforeAll` of the generated suite for the SUT (i.e., *Auth0*) in Figure 2.16.

```
@BeforeEach
fun initTest() {
  controller.resetStateOfSUT()
  wireMock__https__www_doesnotexistfoo_test__6789.resetAll()
  wireMock__https__www_doesnotexistfoo_test__6789.stubFor(
    any(anyUrl())
    .atPriority(100)
    .willReturn(
      aResponse()
        .withHeader("Connection","close")
        .withHeader("Content-Type","text/plain")
        .withStatus(404)
        .withBody("Not Found")
      )
  )
  DnsCacheManipulator.clearDnsCache()
}
```

Figure 2.24: `BeforeEach` of the generated suite for the SUT (i.e., *Auth0*) in Figure 2.16.

we reset the DNS cache to the default configuration using `DnsCacheManipulator`.

Figure 2.25 contains a unit test taken from the generated test suite for the artificial API (i.e., *Auth0* in Figure 2.16). The generated test suite contains all the necessary elements to execute the tests without depending on any external dependencies,

```
1 @Test @Timeout(60)
2 fun test_1()  {
3   DnsCacheManipulator.setDnsCache("www.doesnotexistfoo.test", "127.0.0.22")
4   assertNotNull(wireMock__https__www_doesnotexistfoo_test__6789)
5   wireMock__https__www_doesnotexistfoo_test__6789.stubFor(
6     post(urlEqualTo("/oauth/token"))
7     .atPriority(1)
8     .willReturn(
9       aResponse()
10        .withHeader("Connection","close")
11        .withHeader("Content-Type","application/json")
12        .withStatus(201)
13        .withBody("{" +
14          "\"id_token\": \"W7\"," +
15          "\"refresh_token\": \"KW\"," +
16          "\"expires_in\": -7023511200453181075" +
17          "}"
18        )
19      )
20  )
21
22
23  given().accept("*/*")
24      .header("x-EMextraHeader123", "")
25      .get("${baseUrlOfSut}/api/wm/auth0")
26      .then()
27      .statusCode(200)
28      .assertThat()
29      .contentType("text/plain")
30      .body(containsString("OK"))
31
32
33  wireMock__https__www_doesnotexistfoo_test__6789.resetAll()
34 }
```

Figure 2.25: One unit test taken from the generated test suit for the SUT *Auth0* in Figure 2.16.

as mentioned in Section **??**. In this particular example, the test suite will contain all the necessary configurations to run a mock server successfully without any user interactions. Such generated test suites can be used directly in the development pipelines without any additional steps.

As shown in Figure 2.25, the code initially configures the required DNS information using `DnsCacheManipulator` to redirect the traffic to our mock server (Line 3). Secondly, after the successful assertion of an active mock server, the relevant stubs related to the successful completion of the test will be configured on the respective mock

```
1 @AfterAll
2 @JvmStatic
3 fun tearDown() {
4   controller.stopSut()
5   wireMock__https__www_doesnotexistfoo_test__6789.stop()
6   DnsCacheManipulator.clearDnsCache()
7 }
```

Figure 2.26: `AfterAll` of the generated suite for the SUT (i.e., *Auth0*) in Figure 2.16.

server. In this case, to reach the particular line that returns HTTP 200 (Line 27) with the body `"OK"` (Line 30). Once the necessary assertions are executed, the test will reset the mock server to its default state. Finally, at the end of all test case executions, we make sure that everything shuts down gracefully in a JUnit `AfterAll` call, as shown in Figure 2.26.

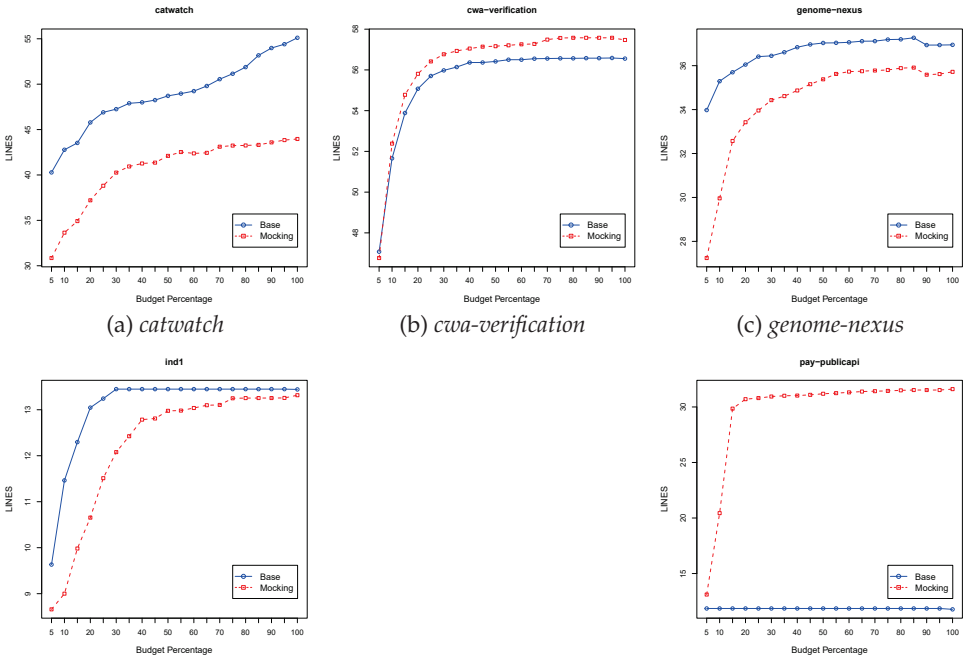
For all these seven artificial APIs, during our experiments, we were able to produce usable test suites. Through our approach, we were able to generate test suites that can cover all the special cases we designed our artificial examples for. With our mocking techniques, full coverage is achieved on all these artificial APIs in a matter of seconds. As all those APIs can now be reliably solved with our novel techniques in a short amount of time, those have been added with E2E tests in the CI build of EvoMaster. All these artificial APIs can be found in the EvoMaster's code repository.[9]

***RQ1**: Our approach, using white-box heuristics and taint analysis, demonstrates its successful capability to guide automatic mock external web service generation for artificial APIs.*

2.7.3 Results for RQ2

To answer **RQ2**, we report line coverage and number of detected faults on average (i.e., *mean*) with 30 repetitions achieved by *Base* and *Mocking*, as shown in Table 2.4. To further demonstrate how the lines are covered throughout the search, we plot the number of covered lines at every 5% intervals of the search budget for each setting in each SUT, as shown in Figure 2.27. The same kinds of graphs for the fault detection criterion are reported in Figure 2.28.

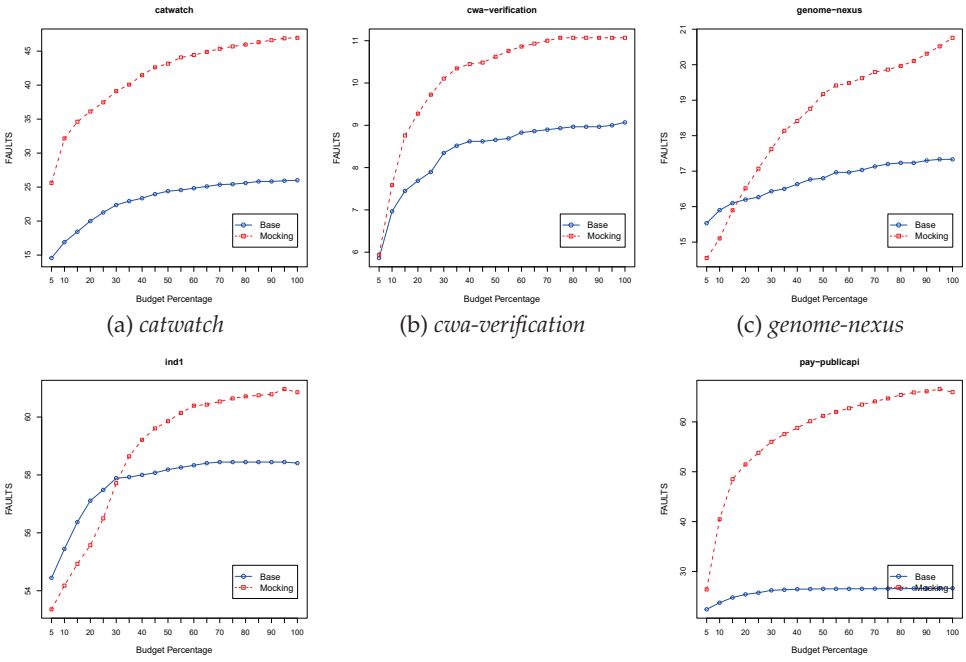
Line coverage is measured with EvoMaster's own bytecode instrumentation. Fault



(a) *catwatch* (b) *cwa-verification* (c) *genome-nexus*

(d) *ind1* (e) *pay-publicapi*

Figure 2.27: The average covered lines (y-axis) with 30 runs achieved by On-*Base* and On-*Mocking* (RQ2), reported at 5% intervals of the budget allocated for the search (x-axis).



(a) *catwatch* (b) *cwa-verification* (c) *genome-nexus*

(d) *ind1* (e) *pay-publicapi*

Figure 2.28: The average number of detected faults (y-axis) with 30 runs achieved by On-*Base* and On-*Mocking* (RQ2), reported at 5% intervals of the budget allocated for the search (x-axis).

Table 2.4: Results of the line coverage and detected faults achieved by *Base* and *Mocking* with internet *On*. We also report pair comparison results between *Base* and *Mocking* using Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) and Mann–Whitney U test (i.e., p-value at significant level 0.05). The number of executed HTTP calls during the search is reported as well.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	Mocking	$\hat{A}_{12}$	p-value	Base	Mocking	$\hat{A}_{12}$	p-value	Base	Mocking	Ratio
catwatch	49.0	47.5	0.40	0.206	25.4	47.0	**1.00**	< 0.001	9412	4047	42.99
cwa-verification	55.6	56.6	**0.78**	< 0.001	9.1	11.1	**0.83**	< 0.001	111346	116944	105.03
genome-nexus	36.9	36.7	**0.31**	0.013	17.3	20.8	**0.98**	< 0.001	17106	27350	159.89
ind1	13.3	13.4	0.58	0.335	57.6	59.3	**0.66**	0.045	109156	108466	99.37
pay-publicapi	11.1	31.8	**1.00**	< 0.001	26.6	40.9	**0.93**	< 0.001	145104	25603	17.64
Average	33.2	37.2	0.61		27.2	35.8	0.88		78425	56482	84.98
Median	36.9	36.7	0.58		25.4	40.9	0.93		109156	27350	99.37

detection is based on the automated oracles used in EvoMaster to detect faults. At the time of these experiments, we used returned 500 HTTP status codes (i.e., server errors) per endpoint as automated oracle. This is a common practice in the research literature on fuzzing REST APIs [25]. No artificial fault was injected in the SUTs. The SUTs used in the experiments are "as-they-are", without modification. All detected faults are actual faults, albeit different faults have different severity and importance [35].

Figure 2.29 contains a test obtained from the generated test suite for *catwatch* to illustrate the reported faults. The selected test simulates a situation in which the external service is available. However, the response body is invalid. In Line 2, using `DnsCacheManipulator`, the traffic to `api.github.com` will be redirected to a loopback address where the respective mock server is running. Similarly, in Line 3, the traffic to `info.services.auth.zalando.com` will be redirected to another mock server. Line 5 assigns the stub to the mock server to return HTTP 200 with an invalid body as the response. Such a response will trigger the application to crash with the error message related to the exception, as seen in Line 33. HTTP 500 is a generic server error message, which is usually returned when there is a crash (e.g., an uncaught thrown exception) in the business logic of the executed endpoint due to a software bug. Generic crashes can be considered as indicative of a fault, as in this case. If the external services an endpoint relies on are either currently unavailable or return invalid data,

```
1 ...
2 DnsCacheManipulator.setDnsCache("api.github.com", "127.0.0.5");
3 DnsCacheManipulator.setDnsCache("info.services.auth.zalando.com", "
    127.0.0.6");
4 assertNotNull(wireMock__https__info_services_auth_zalando_com__443);
5 wireMock__https__info_services_auth_zalando_com__443.stubFor(
6     get(urlEqualTo("/oauth2/tokeninfo"))
7     .atPriority(1)
8     .willReturn(
9         aResponse()
10        .withHeader("Connection","close")
11        .withHeader("Content-Type","application/json")
12        .withStatus(200)
13        .withBody("i6W_KU5cGJM")
14    )
15 );
16
17
18 // Fault100. HTTP Status 500. org/zalando/stups/oauth2/spring/server/
    TokenInfoResourceServerTokenServices_102_getMap GET:/languages
19 // Fault200. Received A Response From API That Is Not Valid According To
    Its Schema. GET:/languages -> Response status 500 not defined for path
    '/languages'.
20 given().accept("application/json")
21 .header("x-EMextraHeader123", "")
22 .get(baseUrlOfSut + "/languages?" +
23     "organizations=_EM_153_XYZ_&" +
24     "limit=341&" +
25     "offset=367&" +
26     "q=_EM_152_XYZ_&" +
27     "access_token=Gt")
28 .then()
29 .statusCode(500)// org/zalando/stups/oauth2/spring/server/
    TokenInfoResourceServerTokenServices_102_getMap
30 .assertThat()
31 .contentType("application/json")
32 .body("'status'", numberMatches(500.0))
33 .body("'error'", containsString("Internal Server Error"))
34 ...
```

Figure 2.29: A generated test for *catwatch*, simulating an application crash scenario, relies on an external web service.

the correct response statuses should either be 503 (Service Unavailable) or 424 (Failed Dependency).

Detailed analyses based on these results are presented next. Before going into those details though, one thing that is important to notice is that the achieved line coverage values reported in Table 2.4 might be lower than what reported in Figure 2.27. This

is not a mistake, and it is actually expected. The values presented in Table 2.4 are after the search is completed, and after a test *minimization* phase is executed. In such a minimization phase, redundant HTTP calls that do not contribute to the achieved coverage are removed. To check this property, test cases must be re-evaluated. Information about the code that is executed only once would be lost, though. This may be due to the fact that static initializers are not considered for coverage metrics in the instrumentation of EvoMaster. However, coverage information about constructors executed in singleton classes (a typical case in Spring and JEE service beans) would be lost as well after this minimization phase. Unfortunately, automatically determining if a constructor belongs to a singleton class is not straightforward. As these issues might only affect reported values in scientific articles, and would have little to no effect to practitioners using EvoMaster, they are not a major concern (a relatively simple fix could be to collect final coverage metrics before the minimization phase).

Based on the analysis of the results reported in Table 2.4, compared to *Base*, we found that *Mocking* achieved significant improvements on both metrics, i.e., line coverage and detected faults. On the one hand, the results show that, regarding fault detection, for all the five selected APIs, our approach (i.e., *Mocking*) significantly outperformed *Base* with low p-values and high $\hat{A}_{12}$. On the other hand, for line coverage, statistically significant improvements were obtained only on two APIs (i.e., *cwa-verification* and *pay-publicapi*), while worse on one (i.e., *genome-nexus*). Those results demonstrate the effectiveness of our approach, which uses white-box heuristics and taint analysis for generating mock external web services.

As we mentioned earlier in Section 4.6.2, *catwatch* relies heavily on GitHub APIs. Due to the rate limiting, we observed impacts on the final results in the line coverage as shown in Figure 2.27. The results for *catwatch* are lower than *Base* (i.e., 49.0%) for *Mocking* (i.e., 47.5%) in line coverage, although such difference is not statistically significant. At the same time, when comparing detected faults, *catwatch* performed better in *Mocking* (i.e., 47.0 faults detected on average) compared to *Base* (i.e., 25.4).

Regarding the *cwa-verification* API, in absolute terms, there is not much difference, only $56.8\% - 55.8\% = 1.0\%pt$ in line coverage. However, such difference is statisti-

```
1 TestResult testResultDob = testResultServerService.result(hashDob);
2
3 // TRS will always respond with a TestResult so we have to check if both
    results are equal
4 if (testResultDob.getTestResult() != testResult.getTestResult()) {
5   // given DOB Hash is invalid
6   throw new VerificationServerException(HttpStatus.FORBIDDEN,
7   "TestResult of dob hash does not equal to TestResult of hash");
8 }
```

Figure 2.30: Code snippet taken from the *cwa-verification*.

cally significant (i.e., p-value < 0.001), with a high value effect-size (i.e., 0.78). We observed similar patterns in the results obtained for **RQ3**, which will be presented in Table 2.5. This is because the *cwa-verification* is part of a microservice architecture, and the interacted service is not up and running on the local machine we used for these experiments. Thus, the two settings for this case study are practically the same.

To get more insight, Figure 2.30 shows a part of the code of *cwa-verification* (code snippet of lines 117–124 in `ExternalTestStateController`[23]). This code can be covered by *Mocking*, but never by *Base* within its 30 repetitions. The code in Figure 2.30 is used to interact with an external web service, to fetch some specific information. Such information will be validated first. Only valid information can be used to implement the business logic of `POST /version/v1/testresult` endpoint in the *cwa-verification* (Lines 4 – 7). As the datatype is clear, when parsing the response (i.e., `TestResult.java`[23]), with our approach (i.e., *Mocking*) the structure of the responses can be easily inferred. During fuzzing, the search can update valid values in the responses to cover more cases in the business logic of the API.

For the *genome-nexus*, we found that most interactions between the SUT and external web services are to fetch data, then saving into a database (i.e., MongoDB). In this case, it is not necessary to extract the data when fetching it from the external web service, rather such data or schema extraction can also be performed when saving data into the database. The code snippet of an external web service interaction and data extraction in *genome-nexus* is shown in Figure 2.31 (the complete code can be found in

```
1 rawValue = this.fetcher.fetchRawValue(this.buildRequestBody(subSet));
2 ...
3 rawValue = this.normalizeResponse(rawValue);
4 ...
5 List<T> fetched = this.transformer.transform(rawValue, this.type);
```

Figure 2.31: Code snippet taken from the *genome-nexus* demonstrating external web service interaction.

`BaseCachedExternalResourceFetcher.java`[23]).

Based on the code in Figure 2.31, a fetch operation is performed initially to get raw data (Line 1). By checking the implementation of `fetchRawValue` (e.g., `BaseExternalResourceFetcher.java`[23]), a general type (such as `BasicDBList`[24]) is provided. With such *Type* information, we can only know it is a list. After that, the actual data extraction based on raw value if performed before saving it into the database (Line 5). Our current handling does not support such multi-level schema extraction of responses yet. This is going to be an important future work. Due to this issue, the line coverage performance of the *genome-nexus* case study shows no improvements, but rather a small, albeit statistically significant, performance loss of 0.2%. On the other hand, we observed considerable performance improvements in detected faults (i.e., 17.3% *vs.* 20.8%). This is likely due to mocking technique enabling testing different error scenarios that are not properly handled by the SUT.

ind1 shows little difference in both metrics. When compared to *Base* (i.e., 13.3%), *Mocking* (i.e., 13.4%) showed a marginal improvement in terms of line coverage. Correspondingly, in detected faults, we can see a small improvement in *Mocking* (i.e., 59.3) compared to *Base* (i.e., 57.6). Due to confidentiality, we are not allowed to share the code of this API. However, it is still important to understand why these kinds of results were obtained. By studying its source code, it seems that most of its endpoints start their execution by calling an external service. But, then, the computation soon crashes due to a null pointer exception (NPE). This is based on a query by resource id to the API's SQL database. The SQL taint analysis used by EvoMaster [10] correctly

[23] `https://github.com/EMResearch/EMB`

[24] `https://mongodb.github.io/mongo-java-driver/3.4/javadoc/com/mongodb/BasicDBList.html`

infers the id of one of the resources used in the provided test data of the API. But, such existing data has some empty columns, whose data will lead to a NPE when used in the code of the SUT. However, no search operators are currently defined in EvoMaster to modify existing SQL test data (can only add new data [10]), nor there is any heuristic in the fitness function to reward queries that return non-null columns. Addressing interactions with external services is only one of the many challenges in fuzzing REST APIs. Other challenges like dealing with SQL databases, although addressed [10], are not fully solved yet, as the case of *ind1* shows. In an API, all these challenges can have unexpected interactions and side-effects.

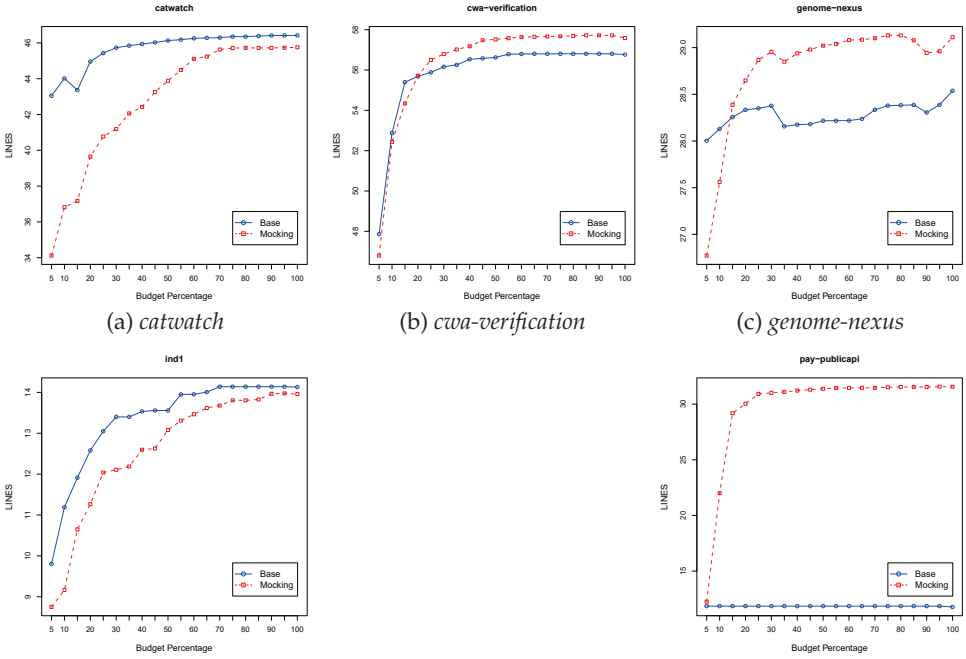
The *pay-publicapi* performed better in both metrics, as the majority of its functionalities are dependent on external web services. When compared with *Base* (i.e., 11.1%), *pay-publicapi* shows a drastic improvement in *Mocking* (i.e., 31.8%, which is more than 20%) in terms of line coverage. Likewise, we observed a similar large improvement in the number of detected faults when comparing *Base* (i.e., 26.6) with *Mocking* (i.e., 40.9).

Overall, compared to *Base*, *Mocking* helped to achieve better results in both metrics as our approach successfully generated mock external web services. On the one hand, the effect is stronger for fault detection, as the use of mocking enables to test some error scenarios that would not be feasible to test with only the real services. On the other hand, the exploration of error scenarios could take time from the search for code coverage, giving some minor side-effects in some cases. But, theoretically, this could change with larger search-budgets (e.g., when fuzzing for more than one hour).

***RQ2**: Our approach with white-box heuristics and taint analysis demonstrates its effectiveness in guiding mock external web service generation, increasing code coverage and fault detection. Such improvements for a search-based fuzzer (i.e., EvoMaster) are statistically significant in most of the five selected APIs. On one API, line coverage improvements were up by +20%.*

2.7.4 Results for RQ3

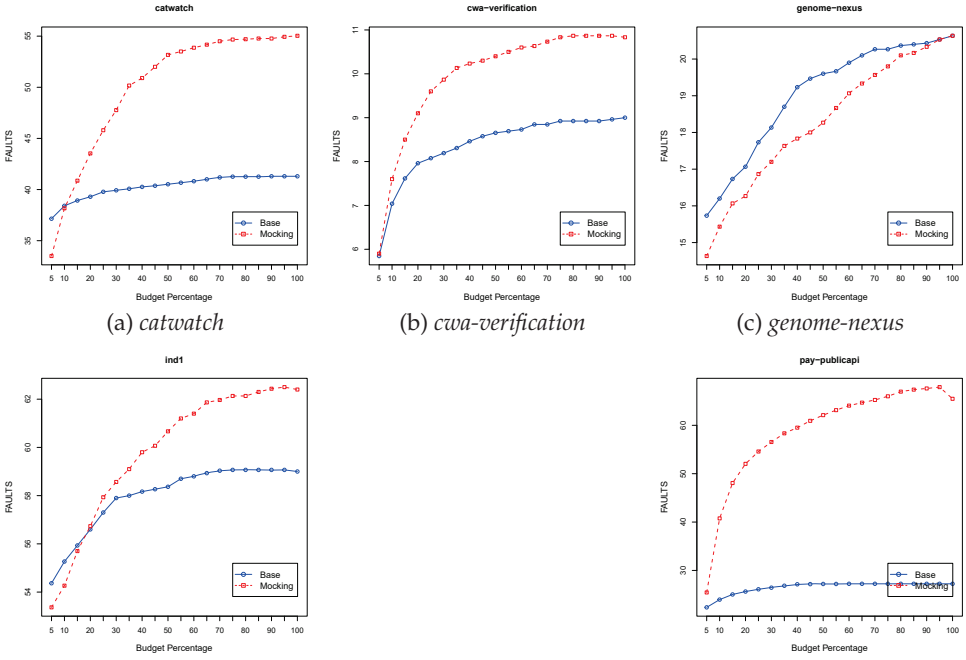
To answer **RQ3**, we ran experiments with 30 repetitions as reported in the Table 2.5. Similar to **RQ2**, we report line coverage and detected faults on average for **RQ3**. We



(a) *catwatch* (b) *cwa-verification* (c) *genome-nexus*

(d) *ind1* (e) *pay-publicapi*

Figure 2.32: Average covered lines (y-axis) with 30 runs achieved by Off-*Base* and Off-*Mocking* (RQ3), reported at 5% intervals of the budget allocated for the search (x-axis).



(a) *catwatch* (b) *cwa-verification* (c) *genome-nexus*

(d) *ind1* (e) *pay-publicapi*

Figure 2.33: The average number of detected faults (y-axis) with 30 runs achieved by Off-*Base* and Off-*Mocking* (RQ3), reported at 5% intervals of the budget allocated for the search (x-axis).

Table 2.5: Results of the line coverage and detected faults achieved by *Base* and *Mocking* with internet *Off*. We also report pair comparison results between *Base* and *Mocking* using Vargha-Delaney effect size (i.e., $\hat{A}_{12}$) and Mann–Whitney U test (i.e., p-value at significant level 0.05). The number of executed HTTP calls during the search is reported as well.

SUT	Line Coverage %				# Detected Faults				# HTTP Calls		
	Base	Mocking	$\hat{A}_{12}$	p-value	Base	Mocking	$\hat{A}_{12}$	p-value	Base	Mocking	Ratio
catwatch	46.3	49.6	**0.73**	0.003	41.3	54.8	**1.00**	< 0.001	26474	25274	95.47
cwa-verification	55.8	56.8	**0.79**	< 0.001	9.0	10.8	**0.86**	< 0.001	113984	121780	106.84
genome-nexus	28.5	30.0	**0.83**	< 0.001	20.6	20.6	0.48	0.761	98623	40852	41.42
ind1	13.9	14.4	**0.70**	0.007	58.2	60.0	**0.79**	< 0.001	124362	102088	82.09
pay-publicapi	11.1	31.6	**1.00**	< 0.001	27.2	41.0	**0.84**	< 0.001	150263	28965	19.28
Average	31.1	36.5	0.81		31.3	37.4	0.79		102741	63792	69.02
Median	28.5	31.6	0.79		27.2	41.0	0.84		113984	40852	82.09

plotted the line coverage throughout the search with 5% intervals for the given budget in Figure 2.32. The same way, we reported the fault detection criterion as graphs in Figure 2.33. Further, we conducted an extended analysis of the results as follows.

Based on the experiment results reported in Table 2.5, compared with *Base*, it can be seen *Mocking* achieved consistent improvements on both metrics, i.e., line coverage and fault detection. Whereas with Internet *On* improvements with line coverage were obtained on only two APIs (recall Table 2.4), here with Internet *Off* line coverage improvements are obtained for all APIs. Interestingly though, in contrast to what is reported in Table 2.4, no improvements for fault detection are detected here in Table 2.5 for *genome-nexus*.

Similar to the results for **RQ2** in Section 2.7.3, with the *catwatch*, we observed significant impacts on the final results of line coverage, as shown as well in Figure 2.27. Line coverage is improved by more than 3% (Table 2.5), and on average 13 more faults are detected. As previously discussed in Section 4.6.2, results for *catwatch* with Internet *On* in Table 2.4 are not reliable, due to the rate limiter imposed by GitHub. Such a rate limiter has no impact on the results reported in Table 2.5.

Regarding *cwa-verification*, such API makes no connections to web services on the internet. Therefore, one would not expect any difference between the results reported in Table 2.4 and in Table 2.5. All the reported small differences can be explained due to

the stochastic nature of repeating experiments 30 times.

Regarding the *genome-nexus*, we observe performance improvements (+1.5%) in line coverage compared to the results in Table 2.4 from **RQ2**. Still, results for *Mocking* are better with Internet *On* (i.e., 36.7 vs. 30.0). This shows the importance of using techniques like Harvester (recall Section 2.5.6) to exploit existing, real data. Regarding the number detected faults, as previously mentioned, there is no difference between *Base* (i.e., 20.6%) and *Mocking* (i.e., 20.6%). This could be explained if all the new detected faults are related to an improper handling of missing connections.

Similar to the results for **RQ2** in Table 2.4, *ind1* shows marginal improvements in both metrics. In line coverage, we observe a slight, but statistically significant, improvement (i.e., 0.5%) in the performance for *Mocking* (i.e., 14.4%) compared with *Base* (i.e., 13.9%). In the same manner, in detected faults *Mocking* (i.e., 60.0%) shows a small increase when compared to *Base* (i.e., 58.2%). For both approaches, we see slightly higher metrics for Internet *Off*. At this point in time, we are not able to provide plausible conjectures to explain this phenomenon.

Similarly, the *pay-publicapi* performed well under both metrics both in internet-enabled experiments (Table 2.4) and internet-disabled experiments (Table 2.5). We observed *Mocking* (i.e., 31.6%) performing well compared to *Base* (i.e., 11.1%) in terms of both line coverage and detected faults (i.e., *Mocking* 41.0% *vs. Base* 27.2%).

It is evident that *Mocking* helped to achieve better results in both metrics compared to *Base*. To summarize, our methodology has successfully generated mock external web services without the necessity of an internet connection.

***RQ3**: Our approach has demonstrated effectiveness in generating mock external web services without an internet connection, increasing code coverage and fault detection. Such improvements for a search-based fuzzer (i.e.,* EvoMaster*) are statistically significant on all the five selected APIs.*

2.7.5 Discussion

Overall, based on the results of our experiments, it is clear that our approach performs well in all seven artificial APIs and in all five selected real-world APIs.

All artificial APIs introduced can be fully covered, reliably, by our novel techniques. As those APIs would be impossible to fully cover without manipulating the external services they connect to, no existing black-box approach [25] would be able to handle them. Therefore, our novel techniques push forward the state-of-the-art in fuzzing Web APIs.

However, results on artificial examples might not necessarily apply to real-world APIs. Our experiments provide scientific evidence to support our claim that our novel techniques provide useful improvements for real-world APIs as well. Results for fault detection were stronger than for line coverage. This can be explained by the fact that, with mocking, we can efficiently test several different error scenarios.

How much improvement can be achieved strongly depends on how reliant the SUT is on communications with external services. If a large API is doing calls to an external service only in one of its endpoints, no many improvements "on average" over the whole API would be achieved, regardless of how good such a case is handled. On the other hand, if the SUT strongly relies on external services, like in the case of *pay-publicapi*, large and substantial improvements can be achieved.

Not all APIs are the same. Handling external services is only one of the current main open problems in fuzzing REST APIs (as identified in [54]). This paper addressed one of the major challenges identified in [54], by providing an automated working solution, implemented in an open-source, state-of-the-art fuzzer, i.e., EvoMaster. Still, there are several research challenges that need to be addressed to further improve the obtained results.

White-box heuristics require instrumentation in the code. This has a computational cost, as more code is executed when evaluating a test case. This is particularly the case when dealing with mocked servers, as those can introduce further computational overheads in their handling. The longer a test case takes to be executed, the fewer number of tests can be evaluated within the same amount of time (e.g., 1-hour fuzzing sessions). Executing fewer test cases means exploring a smaller subset of the search space, which unfortunately could lead to worse results in the end. This is a tradeoff between the quality of a fitness function (e.g., measured in how good it is in "smooth-

ing" the search landscape) and its computational cost (which would reduce the number of fitness evaluations within the same amount of time). Whether it pays-off in the end is not guaranteed.

Our novel techniques presented in this paper are composed of many components that impact the computational cost of each fitness evaluation. Measuring the computational cost of each of them individually would not be viable. However, an indirect measure for performance cost could be to check how many HTTP calls are made during a fuzzing session, given the same amount of time. The higher the computational cost, the fewer calls would be made within the same amount of time. Table 2.4 and Table 2.5 show such data. In cases like *catwatch*, *genome-nexus* and *pay-publicapi*, there is a major drop in number of HTTP calls that are made within an hour. Unfortunately, though, this might not be directly explainable only based on heuristics overhead, as the cost of each fitness function in our context is not constant. For example, a call that returns immediately with a 4xx status code due to invalid inputs would be much quicker to execute compared to a 2xx call that executes large parts of the SUT's business logic, including communications with databases and external services. A novel technique that achieves higher code coverage could evolve test cases that are more computationally expensive to run.

In the end, in our empirical study, this tradeoff paid-off in the end, as even with lower number of HTTP calls, higher code coverage and fault detection were achieved. However, how this would scale to APIs that use a larger number of external communications is something that would be up to empirical investigation.

2.8 Threats to Validity

Internal validity. Our experiment results are derived using a software tool. This can lead to threats to internal validity, due to possible faults in our implemented software. During its development, the software tool was tested using several unit and end-to-end tests (the latter using the existing infrastructure provided by EvoMaster [14]). By contrast, we cannot guarantee that our implementation is fault-free. The tool[9] and the

chosen SUTs[8] used in the experiments are accessible as open-source code on GitHub. Anyone can review them.

Moreover, our experiments are based on a fuzzer that employs some randomness components as part of its internal engine. To take this into account, experiments were repeated 30 times with different initializing random seeds. The results were then analyzed using the appropriate statistical tests, using common practices in the software engineering research literature [6].

In addition, interaction with live web services on the internet are problematic from a research standpoint. They can add noise to the results, especially if they have high variability in the returned HTTP responses, given the same inputs. Although repeating the experiments 30 times helps to reduce the negative impact of such noise on our conclusions, we cannot guarantee that our results can be fully replicated in the future. For example, between the time the experiments were run and the time of reading this text, those web services might no longer exist.

External validity. Our experiments were conducted on five (four open-source and one industrial) selected REST APIs. The four open-source APIs are all the APIs in EMB [16] that deal with connections with external web services. Although enterprise applications typically use external web services, only a few can be found in open-source repositories. Furthermore, running experiments on system-level testing is time-consuming. This limits the number of APIs that can be used for this kind of empirical analysis. Although we used both open-source and industrial APIs, the results of our empirical analysis cannot be generalized at this stage. Therefore, there is a need to investigate more APIs in upcoming studies.

Our novel white-box techniques are implemented for APIs running on the JVM, written for example in either Java or Kotlin. However, there are other popular languages used to build Web APIs, such as JavaScript/TypeScript, C#, Python, Ruby, PHP and Go. Whether the application of our novel techniques on these other languages would require just technical work, or novel research, is something we do not know at the moment. Our previous work on supporting white-box testing in other programming languages would suggest that it should be rather straightforward when dealing

with statically-typed languages (e.g., C# [24]), but there could be few research challenges to address when dealing with dynamically-typed languages (e.g., JavaScript [56]). Regardless, even if our novel techniques would only work on the JVM, this latter is so popular in industry to build enterprise applications (e.g., considering the popularity of frameworks such as SpringBoot[25]) that it could still have meaningful impact on industrial practice.

Our novel techniques have been implemented as an extension to the state-of-the-art white-box fuzzer EvoMaster. Integrating our novel techniques in another evolutionary-based fuzzer should be rather straightforward. However, how to integrate them in other different types of fuzzers is an open research question. Since its inception in 2016, at the time of writing in 2025 EvoMaster is still the only fuzzer for Web APIs (including GraphQL and RPC) that supports white-box fuzzing, albeit the large interest in the research community about fuzzing REST APIs [25] (which so far it has focused on black-box testing). We cannot speculate on how our novel techniques could be integrated in other fuzzers that do not exist, yet.

2.9 Conclusion

In this paper, we have provided a novel search-based approach to enhance white-box fuzzing with automated mocking of external web services. Our techniques have been implemented as an extension of the fuzzer EvoMaster [13], using WireMock for the mocked external services.[9] Experiments on four open-source and one industrial APIs, along with seven artificial APIs show the effectiveness of our novel techniques, both in terms of code coverage and fault detection. On one real-world API (i.e., *pay-publicapi*), line coverage improvements were more than $+20\%$.

To the best of our knowledge, this is the first work in the literature to offer a working solution for tackling this important problem, i.e., how deal with external services in fuzzing of Web APIs. Therefore, there are several avenues for further enhancements of our techniques in future work. An example is how to effectively deal with APIs that

[25] https://theirstack.com/en/technology/spring-boot

have a 2-phase parsing of JSON payloads (like in the case of *genome-nexus* in our case study).

Acknowledgment

This work is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (EAST project, grant agreement No. 864972). Man Zhang is supported by the State Key Laboratory of Complex & Critical Software Environment (CCSE-2024ZX-01).

Bibliography

[1] Afl. https://github.com/google/AFL.

[2] ARCURI, A. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering 23*, 4 (2018), 1959–1981.

[3] ARCURI, A. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology 104* (2018), 195–206.

[4] ARCURI, A. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM) 28*, 1 (2019), 3.

[5] ARCURI, A. Automated black-and white-box testing of restful apis with evomaster. *IEEE Software 38*, 3 (2020), 72–78.

[6] ARCURI, A., AND BRIAND, L. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR) 24*, 3 (2014), 219–250.

[7] ARCURI, A., FRASER, G., AND GALEOTTI, J. P. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering* (2014), pp. 79–90.

[8] ARCURI, A., FRASER, G., AND GALEOTTI, J. P. Generating tcp/udp network data for automated unit test generation. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (2015), pp. 155–165.

[9] ARCURI, A., FRASER, G., AND JUST, R. Private api access and functional mocking in automated unit test generation. In *2017 IEEE international conference on software testing, verification and validation (ICST)* (2017), IEEE, pp. 126–137.

[10] ARCURI, A., AND GALEOTTI, J. P. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM) 29*, 4 (2020), 1–31.

[11] ARCURI, A., AND GALEOTTI, J. P. Enhancing Search-based Testing with Testability Transformations for Existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021), 1–34.

[12] ARCURI, A., AND GALEOTTI, J. P. Enhancing search-based testing with testability transformations for existing apis. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021), 1–34.

[13] ARCURI, A., GALEOTTI, J. P., MARCULESCU, B., AND ZHANG, M. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software 6*, 57 (2021), 2153.

[14] ARCURI, A., ZHANG, M., BELHADI, A., MARCULESCU, B., GOLMOHAMMADI, A., GALEOTTI, J. P., AND SERAN, S. Building an open-source system test generation tool: lessons learned and empirical analyses with evomaster. *Software Quality Journal* (2023), 1–44.

[15] ARCURI, A., ZHANG, M., AND GALEOTTI, J. P. Advanced white-box heuristics for search-based fuzzing of rest apis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2024).

[16] ARCURI, A., ZHANG, M., GOLMOHAMMADI, A., BELHADI, A., GALEOTTI, J. P., MARCULESCU, B., AND SERAN, S. EMB: A curated corpus of web/enterprise ap-

plications and library support for software testing research. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (2023), IEEE, pp. 433–442.

[17] ARCURI, A., ZHANG, M., SERAN, S., GALEOTTI, J. P., GOLMOHAMMADI, A., DUMAN, O., ALDASORO, A., AND GHIANNI, H. Tool report: Evomaster—black and white box search-based fuzzing for rest, graphql and rpc apis. *Automated Software Engineering 32*, 1 (2025), 1–11.

[18] CORRADINI, D., MONTOLLI, Z., PASQUA, M., AND CECCATO, M. Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (2024), pp. 1383–1394.

[19] Dns cache manipulator. https://github.com/alibaba/java-dns-cache-manipulator.

[20] EBERLEIN, M., NOLLER, Y., VOGEL, T., AND GRUNSKE, L. Evolutionary grammar-based fuzzing. In *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12* (2020), Springer, pp. 105–120.

[21] FRASER, G., AND ARCURI, A. EvoSuite: automatic generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)* (2011), pp. 416–419.

[22] GODEFROID, P. Fuzzing: Hack, art, and science. *Communications of the ACM 63*, 2 (2020), 70–76.

[23] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: whitebox fuzzing for security testing. *Communications of the ACM 55*, 3 (2012), 40–44.

[24] GOLMOHAMMADI, A., ZHANG, M., AND ARCURI, A. .NET/C# instrumentation for search-based software testing. *Software Quality Journal* (2023), 1–27.

[25] GOLMOHAMMADI, A., ZHANG, M., AND ARCURI, A. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* (aug 2023).

[26] GOPINATH, R., MATHIS, B., AND ZELLER, A. Mining input grammars from dynamic control flow. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (2020), pp. 172–183.

[27] HARMAN, M., HU, L., HIERONS, R., WEGENER, J., STHAMER, H., BARESEL, A., AND ROPER, M. Testability transformation. *IEEE Transactions on Software Engineering 30*, 1 (2004), 3–16.

[28] HAVRIKOV, N., GAMBI, A., ZELLER, A., ARCURI, A., AND GALEOTTI, J. P. Generating unit tests with structured system interactions. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)* (2017), IEEE, pp. 30–33.

[29] KIM, M., SINHA, S., AND ORSO, A. Adaptive rest api testing with reinforcement learning. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2023), IEEE, pp. 446–458.

[30] KIM, M., XIN, Q., SINHA, S., AND ORSO, A. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2022), ISSTA 2022, Association for Computing Machinery, p. 289–301.

[31] LIU, J., LIU, J., DI, P., LIU, A. X., AND ZHONG, Z. Record and replay of online traffic for microservices with automatic mocking point identification. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (2022), pp. 221–230.

[32] MACKINNON, T., FREEMAN, S., AND CRAIG, P. Endo-testing: unit testing with mock objects. *Extreme programming examined* (2000), 287–301.

[33] MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)* (2012), IEEE, pp. 177–182.

[34] MANÈS, V. J., HAN, H., HAN, C., CHA, S. K., EGELE, M., SCHWARTZ, E. J., AND WOO, M. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering 47*, 11 (2019), 2312–2331.

[35] MARCULESCU, B., ZHANG, M., AND ARCURI, A. On the faults found in rest apis by automated test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 3 (2022), 1–43.

[36] MARRI, M. R., XIE, T., TILLMANN, N., DE HALLEUX, J., AND SCHULTE, W. An empirical study of testing file-system-dependent software with mock objects. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on* (2009), pp. 149–153.

[37] MARTIN-LOPEZ, A., ARCURI, A., SEGURA, S., AND RUIZ-CORTÉS, A. Black-box and white-box test case generation for restful apis: Enemies or allies? In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)* (2021), IEEE, pp. 231–241.

[38] METZMAN, J., SZEKERES, L., SIMON, L., SPRABERY, R., AND ARYA, A. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), pp. 1393–1403.

[39] MILLER, B. P., FREDRIKSEN, L., AND SO, B. An empirical study of the reliability of unix utilities. *Commun. ACM 33*, 12 (dec 1990), 32–44.

[40] MOSTAFA, S., AND WANG, X. An empirical study on the usage of mocking frameworks in software testing. In *2014 14th international conference on quality software* (2014), IEEE, pp. 127–132.

[41] NEUMANN, A., LARANJEIRO, N., AND BERNARDINO, J. An analysis of public rest web service apis. *IEEE Transactions on Services Computing* (2018).

[42] NEWMAN, S. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[43] OLSTHOORN, M., VAN DEURSEN, A., AND PANICHELLA, A. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), pp. 1224–1228.

[44] PARRY, O., KAPFHAMMER, G. M., HILTON, M., AND MCMINN, P. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021), 1–74.

[45] PHAM, V.-T., BÖHME, M., AND ROYCHOUDHURY, A. Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)* (2020), IEEE, pp. 460–465.

[46] RODRÍGUEZ, C., BAEZ, M., DANIEL, F., CASATI, F., TRABUCCO, J. C., CANALI, L., AND PERCANNELLA, G. Rest apis: a large-scale analysis of compliance with principles and best practices. In *International Conference on Web Engineering* (2016), Springer, pp. 21–39.

[47] SAHIN, O., AND AKAY, B. A discrete dynamic artificial bee colony with hyper-scout for restful web service api test suite generation. *Applied Soft Computing 104* (2021), 107246.

[48] SERAN, S., ZHANG, M., AND ARCURI, A. Search-based mock generation of external web service interactions. In *International Symposium on Search Based Software Engineering (SSBSE)* (2023), Springer.

[49] SPADINI, D., ANICHE, M., BRUNTINK, M., AND BACCHELLI, A. To mock or not to mock? an empirical study on mocking practices. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)* (2017), IEEE, pp. 402–412.

[50] STALLENBERG, D., OLSTHOORN, M., AND PANICHELLA, A. Improving test case generation for rest apis through hierarchical clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 117–128.

[51] THOMAS, D., AND HUNT, A. Mock objects. *IEEE Software 19*, 3 (2002), 22–24.

[52] VELDKAMP, L., OLSTHOORN, M., AND PANICHELLA, A. Grammar-based evolutionary fuzzing for json-rpc apis. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)* (2023), IEEE, pp. 33–36.

[53] ZHANG, M., AND ARCURI, A. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021).

[54] ZHANG, M., AND ARCURI, A. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (may 2023).

[55] ZHANG, M., ARCURI, A., LI, Y., LIU, Y., AND XUE, K. White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology 32*, 5 (2023), 1–38.

[56] ZHANG, M., BELHADI, A., AND ARCURI, A. Javascript sbst heuristics to enable effective fuzzing of nodejs web apis. *ACM Transactions on Software Engineering and Methodology* (2023).

[57] ZHANG, M., MARCULESCU, B., AND ARCURI, A. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering 26*, 4 (2021), 1–61.

[58] ZHANG, Y., ZHONG, N., YOU, W., ZOU, Y., JIAN, K., XU, J., SUN, J., LIU, B., AND HUO, W. Ndfuzz: a non-intrusive coverage-guided fuzzing framework for virtualized network devices. *Cybersecurity 5*, 1 (2022), 1–21.

[59] ZHU, X., WEN, S., CAMTEPE, S., AND XIANG, Y. Fuzzing: A survey for roadmap. *ACM Computing Surveys 54*, 11s (sep 2022).

[60] ZIMMERMANN, H. Osi reference model-the iso model of architecture for open systems interconnection. *IEEE Transactions on communications 28*, 4 (1980), 425–432.

Chapter 3

Multi-Phase Taint Analysis for JSON Inference in Search-Based Fuzzing

Seran, Susruthan, Onur Duman, and Andrea Arcuri

SBFT 2025 Search-Based and Fuzz testing

https://doi.org/10.1109/SBFT66712.2025.00015

3.1 Abstract

As software applications grow increasingly complex, particularly in their input formats, testing these applications becomes a challenging endeavour. Automated testing techniques, such as search-based white-box fuzzing, have shown promise in addressing these challenges. However, generating well-formed inputs for fuzzing remains a significant obstacle. In this paper, we present novel techniques as an academic proof-of-concept for automatically inferring JSON-based schemas to enhance search-based white-box fuzzing, focusing on Java and Kotlin applications. Our work offers an alternative approach to black-box grammar-based fuzzing.

3.2 Introduction

Software applications are becoming more and more complex in terms of both their functionality and input formats (e.g., configuration files). Apart from the complex input formats, modern-day web and mobile applications rely on other external web services for several purposes such as payments (e.g., Stripe[1]) and authentication (e.g., OAuth[2]). Typically, this interconnectivity is achieved through web services, and to facilitate such functionalities, the use of third-party Software Development Kits (SDKs) is a widely observed practice in software applications. Due to the versatile nature of the Representational State Transfer (REST) architectural style, the JavaScript Object Notation (JSON) has become the de-facto standard for communication in these web services [25].

Automated software testing techniques (e.g., search-based white-box fuzzing) have proven to be effective approaches to efficiently test such complex applications [36, 13]. However, the generation of well-formed inputs (e.g., based on JSON schemas) for search-based white-box fuzzing is one of several challenges in achieving fully automated testing.

Data Transfer Object (DTO) is a design pattern used in most statically typed pro-

[1] `https://stripe.com/`
[2] `https://oauth.net`

gramming languages (e.g., Java, Kotlin, C++, Go) to represent data, particularly when transferring data over networks or between processes. Typically, DTOs are simple classes that encapsulate data. The usage of such complex input structures is not only common for data transfers in web and mobile applications. For example, there are millions of programs that use *YAML Ain't Markup Language* (YAML) (also known as *Yet Another Markup Language*) configuration files, with Docker[3] being one such example.

User-specified grammars [14] deal with complex inputs for fuzzing. However, using user-specified grammars may not always be feasible. For example, if the System Under Test (SUT) uses an SDK from a vendor, user-specified grammars may not be practical unless the documentation is provided for the SDK. Moreover, to ensure fully automated testing, it is vital to reduce such manual user intervention to near zero.

In this paper, we present novel techniques to infer JSON-based schemas in a fully automated manner for search-based, white-box fuzzing. We developed our techniques as an academic proof-of-concept. Therefore, we will focus on Java and other programming languages that compile into JVM bytecode (e.g., Kotlin), as Java is one of the most widely used programming languages in industry and academia. However, our techniques can be adapted to other statically typed programming languages (e.g., C++, C#, and Go).

In white-box testing, the source code of the SUT needs to be analyzed. The capabilities of the Java Virtual Machine (JVM) allow us to perform this analysis effectively. In particular, Java Reflection and bytecode instrumentation are useful for implementing our techniques. The most popular libraries for JSON parsing are *Gson*[4] and *Jackson*[5] for JVM-based applications [20]. Additionally, our adapted real-world examples are developed using both libraries. As a result, we developed our techniques supporting both libraries.

Existing white-box techniques can handle parsing of JSON data when it is marshaled into DTOs [4, 7]. However, these techniques fail when dealing with parsing container data structures (e.g., maps and lists) when their content-type is not known at

[3] `https://www.docker.com/`
[4] `https://github.com/google/gson`
[5] `https://github.com/FasterXML/jackson`

```
1 public Map<String, String> convertToMap(String token) {
2     Gson gson = new Gson();
3     Map<String, String> tokenMap = new HashMap<String, String>();
4     try {
5         tokenMap = gson.fromJson(token,
6                                  Map.class);
7     }
8     catch (JsonParseException e) {
9         System.out.println("The format of token is invalid. For example {\"
      source1\":\"put-your-token1-here\",\"source2\":\"put-your-token2-here\"}
      ");
10    }
11    return tokenMap;
12 }
13
```

Figure 3.1: Code block from *genome-nexus* responsible for converting *String* into a *Map* of variants.

the time of parsing. In this paper, we provide an automated solution to this problem that is based on a multi-phase taint analysis.

We implemented our techniques as extensions to the open-source white-box fuzzer for Web APIs named EvoMaster,[6] rather than implementing them from scratch, since developing a search-based white-box fuzzer from scratch can be a huge engineering effort. Based on the existing literature, EvoMaster performs best compared to other fuzzers for Web APIs [18, 35, 6].

The rest of this article is organized as follows. First, Section 3.3 provides background information and discusses the motivation behind this work. Second, Section 3.4 presents a literature review. Third, Section 3.5 details our novel techniques. Fourth, Section 3.6 discusses our empirical study to demonstrate the effectiveness of our techniques. After that, Section 3.7 addresses threats to the validity of our work. Finally, Section 3.8 concludes the paper and discusses potential future work.

3.3 Background

In automated testing, there can be cases where inputs are complex and structured

[6] https://github.com/WebFuzzing/EvoMaster/tree/master

rather than simple primitive values (e.g., integer, string, double). To improve the efficiency of testing in terms of code coverage and fault-finding, generating valid inputs is crucial. A study conducted on automated mock generation of external web services observed the impact of not having valid inputs for complex data structures [30]. One way to tackle the challenge of considering such inputs is through grammar-based fuzzing [11]. However, this is not always practical because it may not be always possible to find documentation for third-party SDKs. In addition, it may not always be a scalable approach as the application grows in size (i.e., lines of code). Moreover, generating well-formed inputs in a fully automated manner for testing has not received enough attention in the literature.

To illustrate this issue, we use the code block in Figure 3.1 as a motivating example. Figure 3.1 is taken from one of our SUTs used in our empirical study, which will be presented in Section **??**. In Line 5, the given string input (i.e., the variable `token`) representing a JSON object should be unmarshalled without any failures (i.e., no exceptions are thrown) for the SUT to move further in the execution.

In this example, the SUT is expecting a map of strings as keys and values. Moreover, there can be cases in which the input may contain data structures that are more complex than strings, such as objects or arrays. From our preliminary investigation of the selected case studies, we found more interesting and complex cases of JSON unmarshalling. Additionally, we chose to focus solely on JSON, as it is one of the most common formats for data exchange on the web [24]. Furthermore, compared with grammar-based fuzzing, by focusing solely on JSON, we can perform bytecode analyses that are more efficient and precise than trying to infer input grammars in a black-box or gray-box manner.

3.4 Related Work

3.4.1 Search-based White-Box Fuzzing

Fuzzing is the process of providing random or invalid inputs to software in order to analyze its behavior [15]. Fuzzing is commonly used in the industry to uncover tens of thousands of bugs in large and interconnected enterprise software systems with millions of lines of code [23, 36]. In addition to its usage in industry, fuzzing has received attention in academic research [38, 16]. Fuzzing methods can be categorized as black-box or white-box. In the black-box fuzzing, the tester does not have access to the source code of the application being fuzzed. On the contrary, in white-box fuzzing, the tester has full access to the source code of the application being fuzzed. According to the existing literature, white-box fuzzing is an effective software testing technique [13, 15, 21, 38, 22, 3].

Recently, the use of several fuzzing methods, such as white-box and black-box, to test REST-based web services has gained attention [18, 30, 16]. One common usage of white-box fuzzing is search-based test generation [7, 3, 36, 30, 18]. In search-based test generation, several heuristics can be gathered through white-box fuzzing to improve search performance in terms of code coverage and fault detection rates [7]. Specifically, in an existing study, heuristics about external web services played a key role in improving the quality of test cases generated through automated mocking for search-based test generation [30]. However, the existing literature falls short in terms of addressing the challenges related to heuristics for JSON-based schemas.

3.4.2 Representational State Transfer (REST) and JavaScript Object Notation (JSON)

REST Application Programming Interfaces (APIs) have gradually become the primary means of communication between various services, including microservices and software-as-a-service solutions [12, 25]. REST is a simple and scalable architectural style that uses stateless connections, and this nature makes it suitable for a wide range of use

cases. Based on the existing literature, besides its popularity in industry, REST APIs are one of the popular topics in academia as well [32, 18, 10, 16]. In general, REST APIs use JSON or Extensible Markup Language (XML) as the two common formats to exchange data [24, 3, 28]. JSON is a lightweight data interchange format that supports complex data structures (e.g., arrays and objects) in a key-value pair format and serves as an alternative to XML [1]. In addition to its usage in industry, JSON has received attention in academic research (e.g., [7, 20, 33, 9]).

Additionally, schema specification standards such as OpenAPI [2] are used to create a standardized interface between REST APIs. Such standardization of schemas makes fully automated testing of RESTful APIs feasible [16, 3]. However, for automated search-based fuzzing, it is crucial to infer all schemas used in the SUT to achieve better code coverage. Inferring these complex JSON-based schemas in an automated manner seems to be an area that has received less attention in the existing literature.

3.4.3 Grammars In Fuzzing

Compared with traditional random input-based fuzzing, well-formed inputs in grammar-based fuzzing that use formal grammar specifications enhance the efficiency of the testing process [11, 33]. A combination of search-based testing with grammar-based fuzzing improves the branch coverage for JSON-related classes significantly [27]. In general, such grammars can be user-specified, inferred from documentation (e.g., OpenAPI), or generated using grammar-mining techniques [17]. However, for fully automated and more efficient testing, better grammar-inference techniques are needed.

3.4.4 Taint analysis

Taint analysis is a program analysis technique that tracks data flow within a system, enabling the detection of software faults, especially software vulnerabilities. Taint analysis has been used in various previous studies. Notably, the use of dynamic taint analysis can help to identify security faults with zero-code modifications to the SUT [26]. A study on the detection of sensitive data leakage further demonstrated the viability of

```
1 @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE)
2 public ResponseEntity<String> post(@RequestBody String json) throws
    JsonProcessingException {
3
4     ObjectMapper mapper = new ObjectMapper();
5     Map collection = mapper.readValue(
6             json, Map.class
7     );
8     List<Integer> z = (List<Integer>) collection
9             .get("z");
10
11    if (z.get(1) == 2025) {
12        return ResponseEntity.ok().body("OK");
13    } else {
14        return ResponseEntity.badRequest()
15                .body("FAIL");
16    }
17 }
```

Figure 3.2: Code block of a REST endpoint from one of our artificial case studies.

using taint analysis across different programming languages [34]. A study on the use of taint analysis for advanced white-box heuristics for search-based fuzzing of REST APIs showed significant improvements in testing metrics such as code coverage and fault detection [7]. However, to the best of our knowledge, heuristics for complex, untyped data structure collections (i.e., *Map*, *List*) in search-based testing using dynamic taint analysis have not been presented yet in the existing literature.

3.5 Multi-Phase Taint Analysis

Consider the artificial example in Figure 3.2. Here, an input string is parsed as a JSON element with the Jackson library in Line 5. This JSON string is first marshaled as a `Map`. From this `Map`, the entry `z` is read as a list of integers. On this list, the second element is read and checked if it is equal to 2025. If so, the constant `"OK"` is returned; otherwise, `"FAIL"` is returned. Existing fuzzers (e.g., EvoMaster) would fall short in covering the branch returning `"OK"` since their fitness functions do not provide a gradient (i.e., guidance towards the optimal solution) to the search. By taking this into account, in this paper, we aim to tackle this challenge.

In order to tackle the challenge of covering such cases in testing, given the advanced white-box heuristics based on taint analysis presented in [5, 7], EvoMaster can send tainted values such as `"_EM_k_XYZ_"` (where k is a positive integer, e.g., $k = 42$). Several common APIs are automatically instrumented when the bytecode is first loaded into the JVM of the running SUT [5], such as `readValue()` in *Jackson* [7]. These instrumented calls are semantically equivalent to the original calls (i.e., the SUT must behave the same when it is instrumented) but can analyze if any input is a tainted string. If so, the instrumentation can inform back the search-engine that the input string with the value `"_EM_42_XYZ_"` was attempted to be marshaled into a map object. The next mutation operation on this test case can then replace `"_EM_42_XYZ_"` with a valid map representation, such as `{}`.

This approach works [7], especially well when the marshaled object is a POJO (Plain Old Java Object) or DTO, i.e., a Java class with named fields matching a JSON object representation. For example, consider a JSON object with two fields: `x` (numeric) and `y` (a string), e.g., `{"x":123,"y":"foo"}`. This could be marshaled into a Java class instance having two fields called `x` (of type, for example `double`) and `y` (of type `String`). However, the same JSON object could also be marshaled into a `Map<String,Object>`, where the values `"x"` and `"y"` are string keys in such a map. This is not an unusual case; rather, it is common when there is no POJO/DTO representation for the parsed JSON objects. This can occur, for example, when making a call to an external web service, and the user is interested in only specific fields of the response. Using a `Map` data structure and extracting the needed fields from it may be simpler than implementing a DTO for the entire response. Unfortunately, in this case, when calling `readValue()`, there would be no information on what fields (and their type) would be needed for the remainder of the computation. In other words, by using the techniques in [7], EvoMaster can send string inputs in the form of `"{}"`, but it has no gradient in the search to determine that a field called `z` is needed in the map.

To tackle this first issue, we extend the search-engine of EvoMaster to support the concept of `TaintedMapGene`. When a tainted value `"_EM_k_XYZ_"` is used as input to a marshaling operation for a `Map` in Jackson or Gson, the search process is informed

```
1   L2
2    LINENUMBER 24 L2
3    ALOAD 3
4    LDC "z"
5    INVOKEINTERFACE java/util/Map.get (Ljava/lang/Object;)Ljava/lang/Object
     ; (itf)
6    CHECKCAST java/util/List
7    ASTORE 4
8   L3
9    LINENUMBER 26 L3
10   ALOAD 4
11   ICONST_1
12   INVOKEINTERFACE java/util/List.get (I)Ljava/lang/Object; (itf)
13   CHECKCAST java/lang/Integer
14   INVOKEVIRTUAL java/lang/Integer.intValue ()I
15   SIPUSH 2025
16   IF_ICMPNE L4
```

Figure 3.3: This is a part of the generated bytecode for the example in Figure 3.2.

that the entity holding the tainted value (e.g., a body payload, a query parameter, or field in an object) should be treated as a `TaintedMapGene`. A tainted map will be a immutable value in the form of `{"EM_tainted_map":"_EM_j_XYZ_"}`, for some $j \neq k$, as all tainted values during the search should be unique. Such a map has a specific field, with a tainted string as value. This is a valid map object that would not cause a function such as `mapper.readValue(json, Map.class)` to throw any exceptions. Giving such a map representation as input, we keep track of all key access invocations on the instantiated map (e.g., the variable called `collection` in Figure 3.2). This was already implemented in EvoMaster as part of the advanced branch distance computations for Java collections [5, 7]. For example, every time an instrumented version of `Map.get(keyName)` is called, we check within that instrumented call to see if the map contains any key named `"EM_tainted_map"`.

If so, then we are dealing with a tainted map. Then, the search-engine can be informed that a specific key with value `keyName` (e.g., `"z"`) was accessed on a tainted map with id `"_EM_j_XYZ_"`. With such information, at the next mutation operation, the tainted map will get extended with a new key entry with the name `"z"`.

The second issue is that, although we can infer that the key `"z"` was accessed, we do not know at the time of calling `collection.get("z")` what is the expected type of

`"z"`, as `Map.get(keyName)` returns an instance of the root class `java.lang.Object`. The information about `"z"`, which is expected to be a `List<Integer>` is only obtainable *after* `Map.get(keyName)` is executed. Figure 3.3 shows a snippet of bytecode for a couple of lines of the example in Figure 3.2. After the `Map.get` is called in Line 5 in Figure 3.3, the casting is executed with the bytecode command `CHECKCAST java/util/List`. To discover the information about `"z"`, our solution requires two steps. First, when a new key access (e.g., `"z"`) on a tainted map is discovered, the new entry added to the map by the mutation operation will be of type *String*, with a tainted value. For example, we would end up with something like this: `{"EM_tainted_map":"_EM_j_XYZ_", "z":"_EM_t_XYZ_"}`, for $t \neq j \neq k$ (i.e., all tainted values must be unique). Second, we instrument each call to `CHECKCAST` by adding a probe function before it, in the form of `Object executingCheckCast(Object value, String classType)`. This probe takes as an input the element that is going to be cast and the information on the target cast (`java/util/List` in our example). Such input will be analyzed, and the output will be the input element without modifications (as the probe must not alter the semantics of the function). In this probe, if the input element is a tainted *String* (e.g., `"_EM_t_XYZ_"`), then the search-engine can be informed that such tainted string was tried to be cast into a `List`. At the next mutation operation in the evolutionary process, the entry `"z"` in the tainted map would be replaced with a valid list, e.g., `[]`.

Unfortunately, this is not sufficient, as there is a subtle side effect in the evolutionary process. Before applying our extended taint analysis, an input such as `{"EM_tainted_map":"_EM_j_XYZ_"}` would cause the execution to proceed all the way to `z.get(1)`, throwing a null pointer exception there, as the variable `z` is `null`. A `null` value is a valid `List<Integer>`. On the other hand, an evolved input like `{"EM_tainted_map":"_EM_j_XYZ_", "z":"_EM_t_XYZ_"}` would result in a cast exception being thrown earlier, as a `String` is not a valid `List<Integer>`. This leads to less covered code in the SUT, which in turn results in a worse fitness value. As such, the new evolving test case would die out.

To tackle this third issue, we introduced a new testing target type in EvoMaster.

In particular, in addition to maximizing line and branch coverage (as well as other advanced code-level metrics [5, 7]), we also aim to successfully cover each `CHECKCAST` bytecode instruction with a non-null input. Each `CHECKCAST` instruction will have a fitness score in $[0, 1]$, similar to any other testing target tracked in EvoMaster. The value 0 means that the bytecode instruction was never executed, whereas a 1 means there was at least one non-null input for which the `CHECKCAST` did not throw an exception. Intermediate values are intended to provide gradients for the search. Specifically, a `null` value will be given a non-zero score (e.g., 0.1), while a tainted *String* will get a higher value (e.g., 0.5). This can be computed in the probes added before the `CHECKCAST` instructions. This way, the search has a gradient to reward test cases that are learning the types of accessed fields in the map.

The fourth issue to tackle is that the list itself might be of any type. To handle this case, similar to maps, we introduce the concept of `TaintedArrayGene` in EvoMaster. When we determine that a tainted string is cast into a list or a list is marshaled from a JSON entry, we create a string array with some tainted values. This could be something like `["_EM_a_XYZ_","_EM_a_XYZ_","_EM_a_XYZ_"]`, for $a \neq t \neq j \neq k$. When an element is read from such a list, a cast may take place (see `CHECKCAST java/lang/Integer` in Figure 3.3). In exactly the same way as discussed before for maps, the search-engine can be informed via taint analysis that the elements extracted from this list are cast into integers. At the next mutation operation, the string array `["_EM_a_XYZ_","_EM_a_XYZ_","_EM_a_XYZ_"]` will be mutated into an integer array of random values, e.g., `[42,-5,134234]`.

The last step is to evolve the second element in the array (e.g., -5) into the target value of 2025. To achieve this, we do not need anything novel since we can simply rely on the existing instrumentation for branch-jump instructions (e.g., `IF_ICMPNE`) in EvoMaster, using the traditional branch distance on numerical constraints [19].

By using our novel techniques presented in this paper, it is quite easy for our extended version of EvoMaster to evolve a test case such as the one presented in Figure 3.4, where the `"OK"` branch is covered. Our approach can handle all the different cases in which basic maps and lists/sets/arrays are marshaled from strings represent-

```
@Test @Timeout(60)
public void test_6() {

  given().accept("*/*")
    .header("x-EMextraHeader123", "")
    .contentType("application/json")
    .body(" { " +
                " \"EM_tainted_map\": \"_EM_1897_XYZ_\", " +
                " \"z\": [ " +
                " 268436093, " +
                " 2025, " +
                " -2176 " +
                " ] " +
                " } ")
    .post(baseUrlOfSut+"/api/json")
    .then()
    .statusCode(200)
    .assertThat()
    .contentType("text/plain")
    .body(containsString("OK"));
}
```

Figure 3.4: A generated test for the example mentioned in Figure 3.2.

ing JSON data. Our approach is multi-phase, as each new fitness evaluation uncovers additional information. Then, the discovery of this new information is rewarded in the fitness function, allowing the evolutionary process to continue.

One benefit of this approach is that it is lightweight, as it does not require complex static code analysis. Since it relies on actual input values tracked dynamically through test execution, the control flow of the code has no direct impact on our techniques.

3.6 Empirical Study

To evaluate the novel techniques presented in this paper, we conducted an empirical study to answer the following research question:

RQ: Are our novel techniques capable of automatically inferring JSON-based schemas for case studies adapted from real-world complex examples?

We developed case studies with various business logic adapted from four real-world open-source applications from the EMB corpus [8]. The EMB repository con-

```
1 private List<LocationIQResponse> CallLocationIQAPI(String search) {
2   ArrayList<LocationIQResponse> locationsResponse = new ArrayList<>();
3
4   try {
5       locationsResponse = new ObjectMapper()
6         .readValue(
7             search,
8             new TypeReference<ArrayList<LocationIQResponse>>() {}
9       );
10
11  } catch (IOException e) {
12      LOGGER.error("getLocationIQResponse - IOException - Error with
     message: {}", e.getMessage());
13  }
14
15  return locationsResponse;
16 }
17
```

Figure 3.5: The code block is part of the application logic obtained from the *gestaohospital-rest*.

sists of open-source Web APIs, which have been used by different research groups in various studies [18, 29, 31, 37]. Originally, the adapted examples from the case studies dealt with external web services, database interactions, and messaging brokers. For our experiment purposes, we removed the other factors (i.e., external web service connection and database interactions) and modified the code while preserving the actual logic.

Due to space limitations, we are unable to discuss the case studies we utilized in detail. Therefore, we will provide key details of a few of our case studies in the following steps. At the higher level, all the other case studies have similar elements except the logic behind the JSON unmarshalling.

Figure 3.5 contains the code block adapted from *gestaohospital-rest*. The original code for a service that is responsible for fetching location information from an external web service and converting it into an *ArrayList* of `LocationIQResponse` is shown in Line 6.

To replicate the same behavior as the original application, we created a REST endpoint. Once the conditions are met in Line 9 in Figure 3.6, the endpoint will respond with HTTP 418 and the body `"Tea"` (Line 11); otherwise, it will respond with HTTP

```
1 @PostMapping(path = "/json", consumes = MediaType.APPLICATION_JSON,
    produces = MediaType.APPLICATION_JSON)
2 public ResponseEntity<String> parseJson(@RequestBody String json) {
3   LocationIQService service = new LocationIQService();
4
5   List<LocationIQResponse> responses = service
6         .getLocationIQResponse(json);
7
8   if (responses.get(2).getPlaceId()
9         .equals("teashop")) {
10      return ResponseEntity.status(418)
11        .body("Tea");
12  }
13
14  return ResponseEntity.status(204)
15    .body("No tea for you!");
16 }
17
```

Figure 3.6: The REST endpoint developed as part of *gestaohospital-rest* case study adaptation.

204 and the body `"No tea for you!"`.

Figure 3.7 contains a test from the generated test suite. In Line 7, we can see a successfully generated array of `LocationIQResponse` that allows the application to reach the line of code returning HTTP 418.

Similar to the previous example, Figure 3.8 contains the code obtained from one of the case studies, *proxyprint*.[7] This code is responsible for calculating the budget for already registered print requests. Due to space limitations, we modified and redacted several lines from the original code while preserving the actual logic. The application takes several identifiers of print shops and then calculates the budget based on the data obtained from the database about the shops.

Figure 3.9 contains the code block responsible for parsing JSON inside the `ResultExtender` in the SUT, *languagetool*.[8] This component is responsible for extending the results by adding rule matches from another remote server that contains language rules. Due to this, the original code has an external web service connection. For our purpose, we modified the method argument type from `InputStream` to `String`. In

[7] `https://github.com/ProxyPrint/proxyprint-kitchen`
[8] `https://github.com/languagetool-org/languagetool`

```
@Test @Timeout(60)
fun test_5()  {

    given().accept("application/json")
      .header("x-EMextraHeader123", "")
      .contentType("application/json")
      .body(" [ " +
          " { " +
          " \"placeId\": \"CsUUUU7e8Lpl\" " +
          " }, " +
          " { " +
          " \"placeId\": \"_EM_32_XYZ_\" " +
          " }, " +
          " { " +
          " \"placeId\": \"teashop\" " +
          " } " +
          " ] ")
      .post("${baseUrlOfSut}/api/json?EMextraParam123=_EM_21_XYZ_")
      .then()
      .statusCode(418)
      .assertThat()
      .contentType("application/json")
      .body(containsString("Tea"))
}

```

Figure 3.7: Automatically generated test for *gestaohospital-rest* case study.

```
public Map<Long, String> calcBudgetForPrintRequest(String requestJSON)
    throws IOException {
  PrintRequest printRequest = new PrintRequest();

  List<Long> pshopIDs = null;
  Map prequest = new Gson().fromJson(
    requestJSON, Map.class
  );

  // PrintShops
  List<Double> tmpPshopIDs = (List<Double>) prequest.get("printshops");
  pshopIDs = new ArrayList<>();
  for (double doubleID : tmpPshopIDs) {
    pshopIDs.add((long) Double.valueOf(
      (double) doubleID).intValue()
    );
  }

  // Finally, calculate the budgets :D
  List<PrintShop> pshops = getListOfPrintShops(
    pshopIDs
  );
  Map<Long, String> budgets = printRequest.calcBudgetsForPrintShops(pshops)
    ;

  return budgets;
}
```

Figure 3.8: The application logic obtained from the *proxyprint* case study.

```
private List<RemoteRuleMatch> parseJson(String inputStream) throws
    IOException {
  Map map = mapper.readValue(inputStream, Map.class);
  List matches = (ArrayList) map.get("matches");
  List<RemoteRuleMatch> result = new ArrayList<>();
  for (Object match : matches) {
    RemoteRuleMatch remoteMatch = getMatch(
        (Map<String, Object>) match
    );
    result.add(remoteMatch);
  }
  return result;
}
```

Figure 3.9: The code block demonstrates the application logic obtained from the *languagetool*.

```
1 @PostMapping(path = "/json", consumes = MediaType.APPLICATION_JSON,
    produces = MediaType.APPLICATION_JSON)
2 public ResponseEntity<String> parseJson(@RequestBody String json) {
3   try {
4     ResultExtender resultExtender = new ResultExtender();
5
6     List<RemoteRuleMatch> rules = resultExtender
7         .getExtensionMatches(json);
8
9     if (rules.get(2).getMessage()
10                .equals("vowels")) {
11        return ResponseEntity.ok("vowels");
12    }
13
14    return ResponseEntity.status(204)
15                .body("Nothing found");
16  } catch (IOException e) {
17    return ResponseEntity.status(500).build();
18  }
19 }
```

Figure 3.10: The endpoint developed to adapt the example from the *languagetool* case study.

Line 2, the code unmarshals the string to `Map`. In further steps, obtained data will be extracted through `getMatch()` as in Line 8 and will be added to a new `ArrayList`. Finally, the new `ArrayList` with additional rules will be returned at the end.

In our adaptation, as shown in Figure 3.10, we get the second element from the list and validate that the message string is `"vowels"`. If that's the case, we respond with HTTP 200 and the body `"vowels"`. Otherwise, we respond with HTTP 204 and the body `"Nothing found"`.

Our novel technique automatically produces a usable test suite for all these four examples derived from the EMB repository at the end of the search (i.e., for the endpoints presented in Figure 3.1, 3.2, 3.6, 3.8 and 3.10). Figure 3.11 contains one test from the generated suite for the *proxyprint* case study. Line 7 contains the automatically generated JSON-based schema required for the endpoint to respond with HTTP status code 200. Therefore, we are able to automatically generate an array of `"printshops"` identifiers of the `Long` data type. By taking all of those into account, we can answer the research question as follows.

```
@Test @Timeout(60)
fun test_5()  {

  given().accept("application/json")
  .header("x-EMextraHeader123", "")
  .contentType("application/json")
  .body(" { " +
    " \"EM_tainted_map\": \"_EM_38_XYZ_\", " +
    " \"printshops\": [ " +
    " 3.833483191701392 " +
    " ] " +
    " } ")
  .post("${baseUrlOfSut}/api/json")
  .then()
  .statusCode(200)
  .assertThat()
  .contentType("application/json")
  .body(containsString("Printing"))
}
```

Figure 3.11: A sample of an automatically generated test case for the *proxyprint* case study.

***RQ**: Our approach, which uses white-box heuristics and taint analysis, demonstrates the capability of inferring JSON-based schemas for examples taken from real-world complex APIs.*

3.7 Threats to Validity

Internal validity. Our technique is implemented only for JVM-based applications. There could be other necessary steps required in other programming languages to make our techniques generic. Additionally, our implementation is an extension of EvoMaster. These factors may pose threats to internal validity due to potential faults in our implemented software and the presence of uncovered cases. Furthermore, we cannot ensure that our implementation is free of faults or that our novel techniques can be directly applied to other programming languages. Our extension is open-source, with the link removed due to double-blind reviews.

External validity. Our experiments were conducted on four artificial REST APIs. The four APIs are based and adapted from all the APIs in EMB [8] that have JSON-based

schemas handling. Although we managed to cover several generic scenarios, there could be more edge cases that are not covered under our constructed APIs. Therefore, there is a need to investigate more case studies in future work. Additionally, there is a need to conduct more experiments to study the impact of our approach on testing metrics such as code coverage and fault detection rate on whole, real-world APIs.

3.8 Conclusion

In this paper, we have provided techniques to infer JSON-based schemas for automated search-based white-box fuzzing. Our techniques have been implemented as an extension to the fuzzer EvoMaster [6]. Experiments on four examples based open-source APIs demonstrate the viability of our novel techniques.

To the best of our knowledge, this is the first work in the literature to offer a working solution for tackling this important problem. Therefore, there are several avenues for further enhancement of our techniques in future work. For example, libraries like Jackson[5] use a tree-based data structures (e.g., `JsonNodes`) to represent JSON documents. Those tree-based data structures appear in few APIs in EMB [8]. Our novel techniques do not support such custom representations at the moment.

Acknowledgment

This work is supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

Bibliography

[1] Json schema specification wright draft 00. `https://datatracker.ietf.org/doc/html/draft-wright-json-schema-00`.

[2] Open api specification. `https://swagger.io/specification/`.

[3] ARCURI, A. Automated black-and white-box testing of restful apis with evomaster. *IEEE Software 38*, 3 (2020), 72–78.

[4] ARCURI, A., AND GALEOTTI, J. P. Enhancing Search-based Testing with Testability Transformations for Existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021), 1–34.

[5] ARCURI, A., AND GALEOTTI, J. P. Enhancing search-based testing with testability transformations for existing apis. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021), 1–34.

[6] ARCURI, A., GALEOTTI, J. P., MARCULESCU, B., AND ZHANG, M. Evomaster: A search-based system test generation tool. *Journal of Open Source Software 6*, 57 (2021), 2153.

[7] ARCURI, A., ZHANG, M., AND GALEOTTI, J. P. Advanced white-box heuristics for search-based fuzzing of rest apis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2024).

[8] ARCURI, A., ZHANG, M., GOLMOHAMMADI, A., BELHADI, A., GALEOTTI, J. P., MARCULESCU, B., AND SERAN, S. Emb: A curated corpus of web/enterprise applications and library support for software testing research. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)* (2023), IEEE, pp. 433–442.

[9] BALLESTEROS, I., DE BARRIO, L. E. B., FREDLUND, L.-A., AND MARINO, J. Tool demonstration: Testing json web services using jsongen. *biblioteca.sistedes.es* (2018).

[10] BANIAȘ, O., FLOREA, D., GYALAI, R., AND CURIAC, D.-I. Automated specification-based testing of rest apis. *Sensors 21*, 16 (2021), 5375.

[11] EBERLEIN, M., NOLLER, Y., VOGEL, T., AND GRUNSKE, L. Evolutionary grammar-based fuzzing. In *Search-Based Software Engineering: 12th International*

Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12 (2020), Springer, pp. 105–120.

[12] FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[13] GODEFROID, P. Fuzzing: Hack, art, and science. *Communications of the ACM 63*, 2 (2020), 70–76.

[14] GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation* (2008), pp. 206–215.

[15] GODEFROID, P., LEVIN, M. Y., AND MOLNAR, D. Sage: whitebox fuzzing for security testing. *Communications of the ACM 55*, 3 (2012), 40–44.

[16] GOLMOHAMMADI, A., ZHANG, M., AND ARCURI, A. Testing restful apis: A survey. *ACM Transactions on Software Engineering and Methodology* (aug 2023).

[17] GOPINATH, R., MATHIS, B., AND ZELLER, A. Mining input grammars from dynamic control flow. In *Proceedings of the 28th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (2020), pp. 172–183.

[18] KIM, M., XIN, Q., SINHA, S., AND ORSO, A. Automated test generation for rest apis: No time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2022), ISSTA 2022, Association for Computing Machinery, p. 289–301.

[19] KOREL, B. Automated software test data generation. *IEEE Transactions on software engineering 16*, 8 (1990), 870–879.

[20] MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In *2012 Second International Conference on Digital Information*

and Communication Technology and it's Applications (DICTAP) (2012), IEEE, pp. 177–182.

[21] MANÈS, V. J., HAN, H., HAN, C., CHA, S. K., EGELE, M., SCHWARTZ, E. J., AND WOO, M. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering 47*, 11 (2019), 2312–2331.

[22] MARTIN-LOPEZ, A., ARCURI, A., SEGURA, S., AND RUIZ-CORTÉS, A. Black-box and white-box test case generation for restful apis: Enemies or allies? In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)* (2021), IEEE, pp. 231–241.

[23] METZMAN, J., SZEKERES, L., SIMON, L., SPRABERY, R., AND ARYA, A. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2021), pp. 1393–1403.

[24] NEUMANN, A., LARANJEIRO, N., AND BERNARDINO, J. An analysis of public rest web service apis. *IEEE Transactions on Services Computing* (2018).

[25] NEWMAN, S. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[26] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS* (2005), vol. 5, Citeseer, pp. 3–4.

[27] OLSTHOORN, M., VAN DEURSEN, A., AND PANICHELLA, A. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (2020), pp. 1224–1228.

[28] RODRÍGUEZ, C., BAEZ, M., DANIEL, F., CASATI, F., TRABUCCO, J. C., CANALI, L., AND PERCANNELLA, G. Rest apis: a large-scale analysis of compliance with principles and best practices. In *International Conference on Web Engineering* (2016), Springer, pp. 21–39.

[29] SAHIN, O., AND AKAY, B. A discrete dynamic artificial bee colony with hyper-scout for restful web service api test suite generation. *Applied Soft Computing 104* (2021), 107246.

[30] SERAN, S., ZHANG, M., AND ARCURI, A. Search-based mock generation of external web service interactions. In *International Symposium on Search Based Software Engineering (SSBSE)* (2023), Springer.

[31] STALLENBERG, D., OLSTHOORN, M., AND PANICHELLA, A. Improving test case generation for rest apis through hierarchical clustering. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 117–128.

[32] TSAI, C.-H., TSAI, S.-C., AND HUANG, S.-K. Rest api fuzzing by coverage level guided blackbox testing. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)* (2021), IEEE, pp. 291–300.

[33] VELDKAMP, L., OLSTHOORN, M., AND PANICHELLA, A. Grammar-based evolutionary fuzzing for json-rpc apis. In *The 16th International Workshop on Search-Based and Fuzz Testing* (2023), IEEE/ACM.

[34] WANG, C., KO, R., ZHANG, Y., YANG, Y., AND LIN, Z. Taintmini: Detecting flow of sensitive data in mini-programs with static taint analysis. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)* (2023), IEEE, pp. 932–944.

[35] ZHANG, M., AND ARCURI, A. Open problems in fuzzing restful apis: A comparison of tools.

[36] ZHANG, M., ARCURI, A., LI, Y., LIU, Y., AND XUE, K. White-box fuzzing rpc-based apis with evomaster: An industrial case study. *ACM Transactions on Software Engineering and Methodology 32*, 5 (2023), 1–38.

[37] ZHANG, M., MARCULESCU, B., AND ARCURI, A. Resource and dependency based test case generation for restful web services. *Empirical Software Engineering 26*, 4 (2021), 1–61.

[38] ZHU, X., WEN, S., CAMTEPE, S., AND XIANG, Y. Fuzzing: A survey for roadmap. *ACM Computing Surveys 54*, 11s (sep 2022).

Chapter 4

Detecting Server-Side Request Forgery (SSRF) Vulnerabilities In REST API Fuzz Testing

Susruthan Seran, Guru Prasad Bhandari, and Andrea Arcuri

Submitted for SBFT 2026 Search-Based and Fuzz testing

[Unpublished manuscript]

4.1 Abstract

The entangled internet reveals more vulnerable situations in modern interconnected software systems. Among these, Server-Side Request Forgery (SSRF) is one marked as a top ten vulnerability to look for in the OWASP Top 10 rankings. There has been a significant amount of work in academia to detect SSRF.

In this paper, we present our novel techniques for detecting SSRF in REST APIs, utilizing search-based techniques in a fully automated manner. Our work offers a unique language-agnostic approach, which can be adapted into other software fuzzers for both white-box and black-box contexts.

4.2 Introduction

The ever-evolving internet landscape has pushed software systems to become more interconnected, changing the architecture from monoliths to distributed microservices since the advent of the internet. This evolution introduced greater complexity, making it more vulnerable and prone to errors. In parallel, techniques to find such vulnerabilities are also evolving. Modern-day web applications have a significant number of identified weaknesses, including Server-Side Request Forgery (SSRF) [1], which can lead to unauthorized information retrieval or catastrophic failure of the software system.

The inclusion of SSRF as one of the top ten vulnerabilities in the OWASP Top 10 2021 is an indication of the gravity of the impact [2]. SSRF appears as well in the top 10 vulnerabilities for APIs (OWASP 2023 [3]). For example, in a reported Common Vulnerability Exposure (CVE-2024-35451) [4] related to the LinkStack [1] shows that a simple SSRF could lead to other Common Weakness Enumeration (CWE) linked to Remote Code Execution (RCE) [5, 6]. Recognising the potential impact of SSRF, several attempts have been made to detect it in advance, both in academia and the industry.

In this paper, we present a novel technique for detecting standard SSRF using a

[1] https://linkstack.org/

fully automated search-based approach in both white-box and black-box contexts. We developed our techniques as an academic proof-of-concept. Therefore, we will focus on detecting *standard* and *blind* SSRFs, but leaving the *second-order* SSRF for future research. Furthermore, implementing a fuzzer is a major engineering endeavour; due to this, we extended an existing open-source, state-of-the-art fuzzer named EvoMaster[2] rather than implementing a new fuzzer from scratch. Based on the existing literature, EvoMaster performs best compared to other fuzzers for Web APIs [7, 8, 9, 10]. However, our techniques can be adapted to other existing REST API fuzzers, as they are not tailored to EvoMaster.

Experiments are carried out on 4 artificial APIs and 2 real-world APIs with known SSRF vulnerabilities. In all cases, our novel techniques were able to automatically detect these SSRF vulnerabilities. Furthermore, our techniques are able to generate executable test cases in JUnit format that can reproduce the SSRF exploits, where a local (mock) server is automatically instantiated by the generated test cases themselves to show the malicious connections to it made by the tested APIs.

The remainder of this article is structured as follows. First, Section 4.3 provides background information and outlines the motivation for this research. Following that, Section 4.4 then presents a review of relevant literature. Section 4.5 details our novel techniques. Section 4.6 discusses the empirical study conducted to evaluate the effectiveness of our techniques. Section 4.7 addresses potential threats to the validity of this work. Finally, Section 4.8 concludes the paper and discusses directions for future research.

4.3 Background and Motivation

4.3.1 Server-Side Request Forgery (SSRF)

Figure 4.1 contains a code snippet taken from one of our real-world case studies with reported CVE, Lychee.[3] This was part of a function that allows the user to import

[2] https://github.com/WebFuzzing/EvoMaster/tree/master
[3] https://lycheeorg.dev/

```
1 // If the component parameter is specified, this function returns a string
    (or int in case of PHP_URL_PORT)
2 /** @var string $path */
3 $path = parse_url($url, PHP_URL_PATH);
4 $extension = '.' . pathinfo($path, PATHINFO_EXTENSION);
5
6 if ($extension !== '.') {
7     // Validate photo extension even when `$create->add()` will do later.
8     // This prevents us from downloading unsupported files.
9     BaseMediaFile::assertIsSupportedOrAcceptedFileExtension($extension);
10 }
11
12 // Download file
13 $downloaded_file = new DownloadedFile($url);
14
```

Figure 4.1: Code snippet taken from one our real-world case study, *Lychee*.

images from external sources. The application checks whether the provided URL is an image using the file extension as the indicator in Line 9. However, during our manual inspection, we successfully downloaded a PHP script from our remote server masquerading as a JPG file through the function in Line 13, and the downloaded file is also publicly accessible. The application has a request rewrite rule set for *nginx*,[4] which prevents further exploitation in this case. In addition, we were able to call other services running on the same machine and on the same network through this vulnerability in the application, but our ability to exploit it further was limited by the application's functionality on the affected function.

In the following subsections, we provide a detailed discussion of the common SSRF attacks.

4.3.2 Standard SSRF Attacks

In typical Server-Side Request Forgery (SSRF) attacks, vulnerabilities in an application are exploited to make the application send requests to external servers within the same network, or to internal resources such as files or services located on the same host behind a firewall. This could lead to information retrieval from resources behind the

[4] https://nginx.org/

firewall, or downloading a malicious file from an attacker-controlled server, or even, in some cases, executing malicious commands.

Accessing resources on the internet from user's provided URLs can be part of the normal, expected behavior of the API. However, such URLs must be verified, e.g., to avoid making calls to local servers such as `localhost` and `127.0.0.1` (or `host.docker.internal` if the API is running inside Docker), and to avoid inadvertently enable local file access via `file://` URL protocol. Failure to validate the provided URLs can lead to SSRF exploits.

4.3.3 Blind SSRF Attacks

In blind SSRF, an attacker manipulates a vulnerable application to initiate a request to an unauthorized location; however, the response from the remote server is not relayed to the attacker. As a result, blind SSRF is typically more difficult to detect and exploit than standard SSRF if one does not have direct access to the contacted server.

4.3.4 Second-order SSRF Attacks

Similar to blind SSRF mentioned in Section 4.3.3, second-order SSRF attacks are also harder to detect. In second-order SSRF attacks, the malicious URL is stored (e.g., in a database) when received, and not used immediately. Following calls to other endpoints, or running background jobs, might retrieve such malicious URL at a later time, and use it, leading to an SSRF vulnerability.

4.3.5 Uncommon Attack Surfaces

In most cases, identifying attack surfaces that may result in standard SSRF or other SSRF variants is straightforward, as these typically involve parameters that accept URLs as input. However, there are some unusual attack surfaces in the application that can be used for SSRF attacks.

For example, URLs embedded in data exchange formats such as XML can introduce

vulnerabilities. Parsing XML documents may result in XML External Entity (XXE) attacks, which in turn can expose the application to SSRF.

Additionally, some applications process the HTTP Referer header for analytics purposes. Attackers can manipulate this header to exploit the application for SSRF.

4.4 Related Work

There are several proposed methods in the academic literature with regard to automated vulnerability discovery [11, 12, 13, 14, 15]. Instead of discussing all related works on SSRF, we focus on automated SSRF detection for the Representational State Transfer (REST) Application Programming Interfaces (API). Other contexts, like for example frontend web applications, are not in scope of this paper. For example, the study [12] presented a technique to detect SSRF using dynamic tainting in a PHP web applications. However, the study focuses only on PHP applications having a web GUI frontend.

To the best of our knowledge, there are only two existing approaches in the research literature close to the work presented in this paper in terms of detecting SSRF for REST APIs [15, 11].

The work in [15] presents a general approach to test all different kinds of web APIs, and not just REST, stating it can detect SSRF vulnerabilities as well. But no information on how such checks are done, and how/if they are automated, is provided in [15]. As the tool is no longer available (referenced link[5] on GitHub seems broken), it is not possible to verify what is done for SSRF by checking its source code.

Another study on detecting security vulnerabilities in REST APIs [11] presents an approach that can detect SSRF. However, such an approach is not fully automated, as it requires to incorporate an expert review step in the process to manually verify if the SSRF exploits were successful.

Existing work falls short in terms of a fully automated, language-agnostic approach for SSRF detection in REST APIs. In contrast, the work presented in this paper provides

[5] https://github.com/apif-tool/APIF_tool_2024

a fully automated solution, which is able to generate executable test cases that can reproduce and verify the presence of SSRF vulnerability (if any is present in the tested API).

4.5 Automated SSRF Detection

In this section, we discuss our novel techniques to detect SSRF for REST APIs using a fully automated approach. Our novel SSRF detection techniques are implemented in the search-based fuzzer EvoMaster, but any other fuzzer could had been used. The detection process is divided into five stages: *fuzzing*, *selection*, *identification*, *execution* and *verification*.

In the *fuzzing* phase, the fuzzer is run as usual, aiming at generating a final, small test suite with high coverage (e.g., HTTP coverage and code coverage) and functional fault detection. In the fuzzing literature, this is typically run with search budgets such as 1 hour or 24 hours. The point here is that, typically, generating test cases for happy-day scenarios (e.g., returning HTTP status code in the 2xx family) is a complex task. There might be several constrains in the inputs that could lead to user errors (e.g., HTTP 400 status code). If the software code dealing with external service calls (which might be susceptible to SSRF exploits) is never executed because input validation fails, then there would be no possible effect when sending malicious SSRF payloads. Sending malicious SSRF payloads during the fuzzing session would likely be inefficient. Our approach is to first generate high coverage tests and, then, use those tests as starting templates to apply SSRF attacks.

In the *selection* phase, once the main fuzzing phase is over, we extract test cases from the final test suite that meet specific criteria. For each endpoint defined in the OpenAPI schema of the API, we extract one test case that has at least one HTTP call toward that endpoint (recall a test case can be composed of several HTTP calls), and that has as response a status code in the HTTP 2xx family. If more than one test case fits such criteria, we select the shortest in terms of the number of HTTP calls in it. In case of ties (i.e., same min length), we pick randomly. If for any reason, no test case

is found with a 2xx, we look at test cases that return 400 (generic user error) and 422 (unprocessable content). Likely, those calls have fewer chances to lead to execute the code susceptible to SSRF, but it is not impossible (as we have seen in some preliminary study). However, likely there would be no much point in trying with other 4xx calls that result for example in 401 (not authenticated), 403 (not authorized) or 404 (not found).

After the test case selection, we proceed to the *identification* stage, where the selected test cases for each endpoint are used to identify calls with string input parameters that are likely to take a URL as a value. Trying SSRF payloads on every possible string inputs would be inefficient. So we use a strategy to concentrate only on the input parameters that are likely treated by the API as URLs. To determine if the input can be a URL, we use the parameter name and its description as provided in the SUT's OpenAPI specification. We analysed the corpus from APIs Guru[6] and selected input names that are indicated to be URLs. Using this corpus, we created regular expression (regex) patterns to identify potential inputs from the parameter name and description. In the next step, we use a deterministic approach, utilising the created regex patterns, to classify the input parameters.

In the *execution* phase, we filter all the selected test cases that have at least one identified input parameter as a potential URL. We create clones/duplicates of the test cases for each identified URL parameter. Cloning is done to avoid any negative impact on the existing search results (i.e., the final output test suite). In the cloned test cases, we select only the parameters identified as potentially being URLs. For each of these cloned tests, we update the value of the identified URL parameter with a SSRF payload. The SSRF payload is a HTTP callback URL on the same localhost. Hereafter, we compute the fitness of this cloned and modified test case by executing it. In the next phase, if a HTTP call has parameters marked as potential URLs and could potentially lead to an SSRF fault, it will also be flagged as finding an SSRF fault.

In the *verification* phase, when the cloned tests with SSRF payloads are executed, to validate the vulnerable endpoints, we need to automatically determine if the SSRF

[6] `https://apis.guru/`

exploit was successful. In our approach, we run a dynamically configurable mock HTTP server. Instead of writing our own mock HTTP server, we employed WireMock[7] for this purpose. We created a dynamically configurable `HTTPCallbackVerifier`, where the SSRF payloads point to it. Once a test case with a SSRF payload is executed, if any call was made toward the WireMock instance, then the SSRF exploit was successful. If so, a SSRF fault is found, and these cloned/modified tests are added to the final test suite generated as output of the fuzzer.

Upon completion of this process, test suites are generated (e.g., in JUnit format) with WireMock instantiated and a self-contained `HTTPCallbackVerifier` to automatically validate the exploitability of SSRF.

Furthermore, this approach might enable us to further expand this functionality, allowing us to test for *chained* attacks through SSRF. We leave this for our future work, and we briefly discussed this possibility under Section 4.8.

An important clarification here is that our approach is for *security testing* of REST APIs, and not for *hacking*. To automatically verify that a SSRF was successful, and to generate executable test cases (e.g., in JUnit format) that can flag such issue and be able to reproduce it, the fuzzer has to run on the same network of the tested API. This is not a problem when the user of the fuzzer is a developer of the API (e.g., running the fuzzer in CI [16]) or a QA engineer inside an enterprise tasked to create test cases for the APIs developed there [17]. However, our approach would not directly work when hacking an external API on the internet, without direct access to its local network. On the one hand, a hacker that does a SSRF attack would then need to find a way to detect if it was successful and a way to exploit it. On the other hand, a security tester just needs to discover that an endpoint is vulnerable to SSRF, so that such security fault can be fixed before a way to exploit it maliciously is found.

[7] `https://wiremock.org/`

Table 4.1: Descriptive statistics of the employed synthetic SUTs. For each *SUT*, *#Endpoints* represents the number of declared HTTP endpoints in those APIs. *#Path* represents the source code path in the E2E test folder of our extension of EvoMaster.[2]

SUT	#Endpoints	#Path
Base	1	com/foo/rest/examples/spring/openapi/v3/security/ssrf/base
Header	1	com/foo/rest/examples/spring/openapi/v3/security/ssrf/header
Query	1	com/foo/rest/examples/spring/openapi/v3/security/ssrf/query

4.6 Empirical Study

To evaluate our novel techniques presented in this paper, we conducted an empirical study to answer the following research questions:

RQ1: Can our novel techniques automatically detect SSRF security faults in synthetic APIs?

RQ2: Can our novel techniques automatically detect SSRF security faults in reported vulnerabilities from real-world APIs?

To answer our two research questions, we carried out two different sets of experiments, one per research question, as shown in Table 4.1 and Table 4.2.

4.6.1 Synthetic APIs

To answer **RQ1**, the first set of SUTs consists of seven synthetic REST APIs, developed by us, as listed in Table 4.1. This set includes REST APIs that encompass three distinct SSRF scenarios. The *Base* application addresses standard server-side request forgery in the HTTP request body parameter. This application receives a URL and issues a request to retrieve remote data. Similar to *Base*, the *Query* application also performs similar actions while the vulnerable parameter is in the HTTP request query. In contrast to the other two applications, the *Header* application does not return any information in response to the attacker's request. Instead, it uses the value in the HTTP Referer header to make a call to the remote location, simulating crawling for analytics.

```
1 httpCallbackVerifier55528.resetAll()
2 httpCallbackVerifier55528.stubFor(
3     get("/EM_SSRF_0")
4         .withMetadata(Metadata.metadata().attr("ssrf", "POST:/api/fetch"))
5         .atPriority(1)
6         .willReturn(
7             aResponse()
8                 .withStatus(200)
9                 .withBody("SSRF")
10        )
11 )
12
13 // Verifying that there are no requests made to HttpCallbackVerifier before
       test execution.
14 assertFalse(httpCallbackVerifier55528
15     .allServeEvents
16     .filter { it.wasMatched && it.stubMapping.metadata != null }
17     .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/fetch" }
18 )
19
```

Figure 4.2: Code snippet of setting up the `HttpCallbackVerifier` and the mock server taken from the generated test suite for the synthetic case study, *Base*.

In all three cases, by employing our novel techniques, we were able to produce tests that detect the fault and provide evidence of exploitation.

Figure 4.2 presents the beginning of a test selected from the generated test suite for the synthetic case study *Base*. As the first steps, `HttpCallbackVerifier` will be reset to ensure the verifier is clear in terms of served requests before the actual call made to the SUT. After that, in Line 14, an assertion is performed to make sure that there are no requests made by the SUT and served related to the `RestCallAction`. Further down on the execution, the stub for the request payload will be configured in the mock server with a unique value in the URL path, as in Line 2. This configuration enables identification of the specific request associated with the vulnerable endpoint when the application interacts with the mock server.

In Figure 4.3 and in Line 9, the test suite initiates a call to the vulnerable endpoint and verifies the response in subsequent steps.

Finally, as shown in Figure 4.4 in Line 2, an assertion is performed to ensure that the request has been made to the `HttpCallbackVerifier`, thereby proving the ex-

```
// Fault202. Server-Side Request Forgery (SSRF). sensorUrl.
given().accept("*/*")
        .header("x-EMextraHeader123", "")
        .contentType("application/json")
        .body(" { " +
            " \"sensorUrl\": \"http://localhost:55528/EM_SSRF_0\" " +
            " } ")
        .post("${baseUrlOfSut}/api/fetch")
        .then()
        .statusCode(200)
        .assertThat()
        .contentType("text/plain")
        .body(containsString("OK"))

```

Figure 4.3: Code snippet demonstrates the call made to the vulnerable endpoint in *Base*.

```
// Verifying that the request is successfully made to HttpCallbackVerifier
    after test execution.
assertTrue(httpCallbackVerifier55528
    .allServeEvents
    .filter { it.wasMatched && it.stubMapping.metadata != null }
    .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/fetch" }
)

```

Figure 4.4: The final assertion to ensure the exploitability of SSRF in the vulnerable endpoint in *Base*.

Table 4.2: Descriptive statistics of the employed SUTs. For each *SUT*, where *Version* represents the affected version of the software, where *CVE* represents the issued CVE number. Finally, *Endpoint* represents the affected REST endpoint.

SUT	Version	CVE	Endpoint
microcks	1.7.1	CVE-2023-48910	/artifact/download
lychee	6.6.13	CVE-2025-53018	/api/v2/Photo::fromUrl

istence of SSRF and exploitability.

In addition to the listed synthetic APIs in Table 2.2, we developed another open-source case study to test our novel techniques in black-box settings. *Clerk*[8] is an open-source REST API being developed to demonstrate software vulnerabilities in a real-world context, that can be used for testing and education purposes. We used this open-source API (having 12 endpoints), to study whether our techniques would work in more complex scenarios. Figure 4.5 contains the generated test for *Clerk* in black-box, illustrating the whole test execution flow. We were to detect SSRF in the endpoint to fetch the product image from a remote location. Apart from the reported steps in the previous examples, as in Figure 4.5, Line 4, we report additional information such as the fault category reported in the generated tests.

***RQ1**: Our approach demonstrates the capability of automatically detecting SSRF faults for synthetic examples.*

4.6.2 Real-World APIs

To answer **RQ2**, we decided to evaluate our approach using real-world applications. Therefore, our second set of SUTs consists of four APIs that have reported CVEs, as selected from the CVE database.[9]

microcks is a platform for turning your API and microservices assets - *OpenAPI specs*, *AsyncAPI specs*, *gRPC protobuf*, *GraphQL schema*, *Postman collections*, *SoapUI projects* - into live mocks in seconds.[10] Note that, on the original reporting of the vulnerability

[8] `https://github.com/redacted/redacted`
[9] `https://www.cve.org`
[10] `https://microcks.io`

in the CVE database, the version was wrongly reported (i.e., 1.17.1 instead of 1.7.1).

lychee is a free, open-source photo-management PHP tool that runs on your server or webspace.[11]

Given the stochastic behaviour of search algorithms, we repeated each experimental setting 10 times under a consistent termination criterion of 1 hour, in accordance with established guidelines for evaluating randomised techniques in software engineering research [18]. In this context, EvoMaster was run in black-box mode (i.e., as white-box mode only works for JVM APIs, and *Lychee* is written in PHP). Test cases with SSRF exploits were successfully generated for both case studies. For *Lychee*,[11] SSRF was detected and exploited in two out of ten runs. Similarly, for *Microcks*,[10] SSRF was detected and exploited in six out of ten runs.

Figure 4.6 contains a part of the test taken from the generated test suites for Lychee with SSRF detection. This tests the functionality shown in Figure 4.1 in Section 4.3. This endpoint accepts multiple URLs as an array, resulting in repeated values. At the end of the assertion, the SUT returns HTTP 422 because the application expects an image on the given URL. Additionally, the album identifier is required in the request for the application to process the images further. If the provided album identifier is invalid, the application returns an HTTP 422 before processing the URLs. On the contrary, if the album identifier is `null`, the application tries to download the images. In this example, since the request made in the test shown in Figure 4.6, the provided URL does not have an image, and the album identifier is `null`, the application returns HTTP 422.

In the end, as shown in Figure 4.7, we were able to validate the detected SSRF in this endpoint. Because the application has a validation rule, if there is an album identifier in the request, the SSRF detection process exhibits a low success rate.

Meanwhile, in *Microcks*,[10] similar to *Lychee*, we ran experiments 10 times under a consistent termination criterion of 1 hour. The vulnerable endpoint takes a URL as one parameter, and if the parameter is valid, the application processes the request; otherwise, it responds with HTTP 500. Since we do not consider HTTP 500 during the selection stage for SSRF detection, this created an impact on the success rate. Compared

[11] `https://lycheeorg.dev`

with *Lychee*, *Microcks* performed better in our experiments due to the less complexity in the affected endpoint.

Overall, our approach effectively identified SSRF in both real-world case studies. However, in contrast to the experiments on synthetic APIs, SSRF vulnerabilities were not detected in all the experiment repetitions on real-world APIs. Real-world APIs might have complex input and state validation. Even when using state-of-the-art fuzzers such as EvoMaster, it does not mean that the right input data can be generated reliably each time.

***RQ2**: Our approach demonstrates the capability of automatically detecting SSRF faults for real-world applications which have reported CVEs for SSRF.*

4.7 Threats To Validity

Internal validity. The experiments were conducted primarily using a software tool, from which the results were derived. Software faults are inevitable and may introduce threats to internal validity. Moreover, during the implementation, the software tool was tested extensively at various levels (i.e., unit and end-to-end tests provided by EvoMaster [19]). However, it cannot be guaranteed that the implementation is entirely free of faults. Our implementation is built on top of the open-source fuzzer EvoMaster [19]. The extended tool used in this study and the example APIs are accessible as open-source code on GitHub.[12] Anyone can review them.

External validity. The experiments were conducted on three synthetic example APIs, one artificial open-source didactic project, and against two real-world APIs with previously disclosed CVEs for SSRF. The three synthetic APIs are modeled on real-world examples adapted from some reported Common Vulnerabilities Exposures. Although common scenarios from real-world applications were considered, it is likely that some edge cases remain unaddressed. Further investigation is required to address additional cases, which are identified as future work.

[12]link removed due to double-blind reviews

4.8 Conclusions

In this paper, we have provided novel techniques to detect SSRF using a fully automated approach that is adaptable in both white-box and black-box fuzzing strategies. Our techniques have been implemented as an extension to the fuzzer EvoMaster [20]. Experiments on 6 REST APIs demonstrate the viability of our language-agnostic novel techniques.

There are several avenues for further enhancement of our techniques in future work. The second-order SSRF mentioned in Section 4.3 detection is a complex issue that varies depending on the application. Furthermore, building on this approach of using a dynamic input, such as the use of a configurable `HttpCallbackVerifier`, there is research potential to create novel techniques to identify other vulnerability classes that involve external connections. Compared with using a static payload, we can expand the search to look for other cases where SSRF could lead to other vulnerabilities, such as SQL injection, command injection, and code injection. For example, in reported CVE (CVE-2024-35451) [4] for *LinkStack* [1] leads to a Command Injection [5] through SSRF.

Acknowledgments

Removed due to double-blind reviews.

Bibliography

[1] MITRE. (2013) CWE-918: Server-Side Request Forgery (SSRF). MITRE Corporation. CWE. [Online]. Available: https://cwe.mitre.org/data/definitions/918.html

[2] OWASP. Owasp top 10 - 2021. [Online]. Available: https://owasp.org/www-project-top-ten/

[3] ——. (2023) About owasp. [Online]. Available: https://owasp.org/API-Security/editions/2023/en/0x01-about-owasp/

[4] NVD. (2024) CVE-2024-35451. National Vulnerability Database. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-35451

[5] MITRE. (2006) CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection'). MITRE Corporation. CWE. [Online]. Available: https://cwe.mitre.org/data/definitions/77.html

[6] ——. (2006) CWE-94: Improper Control of Generation of Code ('Code Injection'). MITRE Corporation. CWE. [Online]. Available: https://cwe.mitre.org/data/definitions/94.html

[7] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated Test Generation for REST APIs: No Time to Rest Yet," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–301. [Online]. Available: https://doi.org/10.1145/3533767.3534401

[8] M. Zhang and A. Arcuri, "Open Problems in Fuzzing RESTful APIs: A Comparison of Tools," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, may 2023. [Online]. Available: https://doi.org/10.1145/3597205

[9] H. Sartaj, S. Ali, and J. M. Gjøby, "Rest api testing in devops: A study on an evolving healthcare iot application," *ACM Transactions on Software Engineering and Methodology*, 2025.

[10] O. Sahin, M. Zhang, and A. Arcuri, "Wfc/wfd: Web fuzzing commons, dataset and guidelines to support experimentation in rest api fuzzing," 2025. [Online]. Available: https://arxiv.org/abs/2509.01612

[11] W. Du, J. Li, Y. Wang, L. Chen, R. Zhao, J. Zhu, Z. Han, Y. Wang, and Z. Xue, "Vulnerability-oriented testing for restful apis," in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024, pp. 739–755.

[12] E. Wang, J. Chen, W. Xie, C. Wang, Y. Gao, Z. Wang, H. Duan, Y. Liu, and B. Wang, "Where urls become weapons: Automated discovery of ssrf vulnerabilities in web applications," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 239–257.

[13] G. Deng, Z. Zhang, Y. Li, Y. Liu, T. Zhang, Y. Liu, G. Yu, and D. Wang, "{NAUTILUS}: Automated {RESTful}{API} vulnerability detection," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5593–5609.

[14] S. Jan, A. Panichella, A. Arcuri, and L. Briand, "Search-based multi-vulnerability testing of xml injections in web applications," *Empirical Software Engineering*, vol. 24, pp. 3696–3729, 2019.

[15] Y. Wang and Y. Xu, "Beyond rest: Introducing apif for comprehensive api vulnerability fuzzing," in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, 2024, pp. 435–449.

[16] A. Arcuri, P. Garrett, J. P. Galeotti, and M. Zhang, "Widening the adoption of web api fuzzing: Docker, github action and python support for evomaster," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, 2025, pp. 1084–1088.

[17] A. Arcuri, A. Poth, and O. Rrjolli, "Introducing black-box fuzz testing for rest apis in industry: Challenges and solutions," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2025.

[18] A. Arcuri and L. Briand, "A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering," *Software Testing, Verification and Reliability (STVR)*, vol. 24, no. 3, pp. 219–250, 2014.

[19] A. Arcuri, M. Zhang, A. Belhadi, B. Marculescu, A. Golmohammadi, J. P. Galeotti, and S. Seran, "Building an open-source system test generation tool: lessons learned and empirical analyses with evomaster," *Software Quality Journal*, pp. 1–44, 2023.

[20] A. Arcuri, J. P. Galeotti, B. Marculescu, and M. Zhang, "EvoMaster: A Search-Based System Test Generation Tool," *Journal of Open Source Software*, vol. 6, no. 57, p. 2153, 2021.

```
/**
* Calls:
* (422) POST:/api/product/fetch/image
* Found 1 potential fault of type-code 202
*/
@Test @Timeout(60)
fun test_7_postOnImageVulnerableToSSRF()  {
    assertNotNull(httpCallbackVerifier59917.isRunning)
    httpCallbackVerifier59917.resetAll()
    httpCallbackVerifier59917.stubFor(
        get("/EM_SSRF_2")
            .withMetadata(Metadata.metadata().attr("ssrf", "POST:/api/
    product/fetch/image"))
            .atPriority(1)
            .willReturn(
                aResponse()
                    .withStatus(200)
                    .withBody("SSRF")
            )
    )
    // Verifying that there are no requests made to HttpCallbackVerifier
    before test execution.
    assertFalse(httpCallbackVerifier59917
        .allServeEvents
        .filter { it.wasMatched && it.stubMapping.metadata != null }
        .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/
    product/fetch/image" }
    )
    // Fault202. Server-Side Request Forgery (SSRF). url.
    given().accept("*/*")
            .contentType("application/json")
            .body(" { " +
                " \"url\": \"http://host.docker.internal:59917/EM_SSRF_2\",
     " +
                " \"productId\": 257 " +
                " } ")
            .post("${baseUrlOfSut}/api/product/fetch/image")
            .then()
            .statusCode(422)
            .assertThat()
            .contentType("text/plain")
            .body(containsString("Unable to fetch."))
    // Verifying that the request is successfully made to
    HttpCallbackVerifier after test execution.
    assertTrue(httpCallbackVerifier59917
        .allServeEvents
        .filter { it.wasMatched && it.stubMapping.metadata != null }
        .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/
    product/fetch/image" }
    )
}
```

Figure 4.5: Code snippet demonstrates the full flow of the text execution of the vulnerable endpoint in *Clerk*, tested under black-box settings.

```
// Fault202. Server-Side Request Forgery (SSRF). urls_item.
given().accept("application/json")
        .header("Authorization", "IoDq6KQsD5atHyBi7sFlbQ==") // Fixed Headers
        .header("X-Requested-With", "XMLHttpRequest") // Fixed Headers
        .header("Content-Type", "application/json") // Fixed Headers
        .contentType("application/json")
        .body(" { " +
            " \"album_id\": \"\", " +
            " \"urls\": [ " +
            " \"http://host.docker.internal:46213/EM_SSRF_0\" " +
            " ] " +
            " } ")
        .post("${baseUrlOfSut}/api/v2/Photo::fromUrl")
        .then()
        .statusCode(422)
        .assertThat()
        .contentType("application/json")
        .body("'message'", containsString("A photo could not be imported"))
        .body("'exception'", containsString("MassImportException"))

```

Figure 4.6: Code snippet illustrates the HTTP request made to the vulnerable endpoint in *Lychee*.

```
// Verifying that the request is successfully made to HttpCallbackVerifier after test execution.
assertTrue(httpCallbackVerifier46213
    .allServeEvents
    .filter { it.wasMatched && it.stubMapping.metadata != null }
    .any { it.stubMapping.metadata.getString("ssrf") == "POST:/api/v2/Photo::fromUrl" }
)

```

Figure 4.7: Code snippet which verifies the successful exploitation of SSRF in *Lychee*.

Kristiania University of Applied Sciences
PO Box 1190 Sentrum
NO-0107 Oslo

PhD Dissertations Kristiania 2026:1
ISSN: 3084-1488 (print)
ISSN: 3084-147X (online)
ISBN: 978-82-93953-14-2 (print)
ISBN: 978-82-93953-13-5 (online)